INTERNATIONAL STANDARD

IEC 60488-2

First edition
2004-05

IEEE 488.2™

Standard digital interface for programmable instrumentation –

Part 2: Codes, formats, protocols and common commands

Reference number
IEC 60488-2(E):2004
IEEE Std. 488.2(E):1992

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site (www.iec.ch)**

- **Catalogue of IEC publications**

  The on-line catalogue on the IEC web site (www.iec.ch/searchpub) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

  This summary of recently issued publications (www.iec.ch/online_news/ justpub) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

  If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

  Email: custserv@iec.ch
  Tel: +41 22 919 02 11
  Fax: +41 22 919 03 00

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

INTERNATIONAL STANDARD

IEC 60488-2

First edition
2004-05

IEEE 488.2™

Standard digital interface for programmable instrumentation –

Part 2: Codes, formats, protocols and common commands

© IEEE 2004 — Copyright - all rights reserved

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Inc.

No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland
Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc, 3 Park Avenue, New York, NY 10016-5997, USA
Telephone: +1 732 562 3800 Telefax: +1 732 562 1571 E-mail: stds-info@ieee.org Web: www.standards.ieee.org

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

CONTENTS

Foreword 6

1. Introduction 10

1.1 Scope 10
1.2 Objectives 11
1.3 Notation 11

2. References 11

3. System Considerations 12

3.1 Definitions 12
3.2 System Message Traffic 13
3.3 Functional Layers 14

4. Device Compliance Criteria 15

4.1 IEEE 488.1 Requirements 16
4.2 Message Exchange Requirements 16
4.3 Syntax Requirements 16
4.4 Status Reporting Requirements 18
4.5 Common Commands 21
4.6 Synchronization Requirements 24
4.7 System Configuration Capability 24
4.8 Controller Capability 25
4.9 Device Documentation Requirements 25

5. Device Interface Function Requirements 26

5.1 Handshake Requirements 26
5.2 Address Requirements 27
5.3 Talker Requirements 27
5.4 Listener Requirements 27
5.5 Service Request Requirements 28
5.6 Remote/Local Requirements 28
5.7 Parallel Poll Requirements 30
5.8 Device Clear Requirements 31
5.9 Device Trigger Requirements 31
5.10 Controller Function Requirements 31
5.11 Electrical Requirements 31
5.12 Power-On Requirements 32

6. Message Exchange Control Protocol 33

6.1 Functional Elements 33
6.2 Protocol Overview 45
6.3 Message Exchange Control Operation 47
6.4 Protocol Rules 52
6.5 Protocol Exceptions 55

7. Device Listening Formats 57

7.1 Overview 57
7.2 Notation 58
7.3 Terminated Program Messages – Functional Syntax 61

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved. ,,-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

7.4 Separator Functional Elements 65
7.5 <PROGRAM MESSAGE TERMINATOR> 67
7.6 Program Header Functional Elements 67
7.7 <PROGRAM DATA> Functional Elements 72

8. Device Talking Elements 87

8.1 Overview 87
8.2 Notation 88
8.3 Terminated Response Messages – Functional Syntax 88
8.4 Separator Functional Elements 92
8.5 <RESPONSE MESSAGE TERMINATOR> 93
8.6 <RESPONSE HEADER> 94
8.7 <RESPONSE DATA> Functional Elements 95

9. Message Data Coding 106

9.1 ASCII 7 bit Codes 107
9.2 Binary 8 Bit Integer Codes 109
9.3 Binary Floating Point Code 110

10. Common Commands and Queries 113

10.1 *AAD, Accept Address Command 116
10.2 *CAL?, Calibration Query 116
10.3 *CLS, Clear Status Command 117
10.4 *DDT, Define Device Trigger Command 118
10.5 *DDT?, Define Device Trigger Query 119
10.6 *DLF, Disable Listener Function Command 120
10.7 *DMC, Define Macro Command 120
10.8 *EMC, Enable Macro Command 123
10.9 *EMC?, Enable Macro Query 124
10.10 *ESE, Standard Event Status Enable Command 125
10.11 *ESE?, Standard Event Status Enable Query 125
10.12 *ESR?, Standard Event Status Register Query 126
10.13 *GMC?, Get Macro Contents Query 126
10.14 *IDN?., Identification Query 127
10.15 *IST?, Individual Status Query 129
10.16 *LMC?, Learn Macro Query 129
10.17 *LRN?, Learn Device Setup Query 130
10.18 *OPC, Operation Complete Command 131
10.19 *OPC?, Operation Complete Query 131
10.20 *OPT?, Option Identification Query 132
10.21 *PCB, Pass Control Back 133
10.22 *PMC, Purge Macros Command 134
10.23 *PRE, Parallel Poll Enable Register Command 134
10.24 *PRE?, Parallel Poll Enable Register Query 135
10.25 *PSC, Power-On Status Clear Command 136
10.26 *PSC?, Power-On Status Clear Query 137
10.27 *PUD, Protected User Data Command 137
10.28 *PUD?, Protected User Data Query 138
10.29 *RCL, Recall Command 139
10.30 *RDT, Resource Description Transfer Command 140

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.31 *RDT?, Resource Description Transfer Query 141
10.32 *RST, Reset Command 142
10.33 *SAV, Save Command 143
10.34 *SRE, Service Request Enable Command 143
10.35 *SRE?, Service Request Enable Query 144
10.36 *STB?, Read Status Byte Query 145
10.37 *TRG, Trigger Command 145
10.38 *TST?, Self-Test Query 146
10.39 *WAI, Wait-to-Continue Command 147
10.40 *RMC, Remove Individual Macro Command 147
10.41 *SDS, Save Default Device Settings Command 148

11. Device Status Reporting 149

11.1 Overview 149
11.2 Status Byte Register 150
11.3 Service Request Enabling 155
11.4 Status Data Structures 161
11.5 Standard Status Data Structure 166
11.6 Parallel Poll Response Handling 173

12. Device/Controller Synchronization Techniques 174

12.1 Overview 174
12.2 Sequential and Overlapped Commands 175
12.3 Pending-Operation Flag 176
12.4 No-Operation-Pending Flag 176
12.5 Controller/Device Synchronization Commands 176
12.6 Synchronization With External-Control-Signals 180
12.7 Improper Usage of *OPC and *OPC? 181
12.8 Design Considerations 182

13. Automatic System Configuration 184

13.1 Introduction 184
13.2 Overview 185
13.3 Generic Approach to Automatic System Configuration 186
13.4 Detailed Requirements of the Auto Configuration Commands 191
13.5 Additional Automatic Configuration Techniques 197
13.6 Examples 197

14. Controller Compliance Criteria 197

14.1 IEEE 488.1 Requirements 197
14.2 Message Exchange Requirements 198
14.3 Protocols 198
14.4 Functional Element Handling 200
14.5 Controller Specification Requirements 200

15. I IEEE 488.2 Controller Requirements 200

15.1 Controller Interface Function Requirements 200
15.2 Additional IEEE 488.2 Controller Requirements 201
15.3 IEEE 488.2 Controller Recommendations 202

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16. Controller Message Exchange Protocols....202

16.1 Definitions....203
16.2 Control Sequences....205

17. Common Controller Protocols....213

17.1 Reset Protocol....214
17.2 Find Device Requesting Service Protocol....215
17.3 Serial Poll All Devices Protocol....216
17.4 Pass Control Protocol....218
17.5 Requesting Control....219
17.6 Find Listeners Protocol....220
17.7 Set Address Protocol....221
17.8 Test System Protocol....226

Annex A Compound Headers – Usage and Examples (Informative)....228

Annex B Device/Controller Synchronization Techniques (Informative)....232

Annex C Automatic System Configuration Example (Informative)....244

Annex D Reset Guidelines (Informative)....250

Annex E List of Participants (Informative)....256

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

INTERNATIONAL ELECTROTECHNICAL COMMISSION

STANDARD DIGITAL INTERFACE FOR PROGRAMMABLE INSTRUMENTATION –

Part 2: Codes, formats, protocols and common commands

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 60488-2 has been processed through subcommittee 65C: Digital communications, of IEC technical committee 65: Industrial-process measurement and control.

This standard cancels and replaces the second edition of IEC 60625-2 (1993).

At times in this standard, specific reference is made to IEEE Std 488.1:1987, which constituted an earlier version of IEEE Std 488.1:2003, the IEEE edition upon which IEC/IEEE 60488-1:2004 is based. Where specific dated references were made to the 1987 edition, these references have been maintained.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
488.2 (1992)	65C/320/FDIS	65C/326/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives.

The committee has decided that the contents of this publication will remain unchanged until 2009.

IEC/IEEE 60488 consists of the following publications:

- Higher performance protocol for the standard digital interface for programmable instrumentation – Part 1: General (60488-1).
- Standard digital interface for programmable instrumentation – Part 2: Codes, formats, protocols and common commands (60488-2).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied “AS IS”.

The existence of an IEC/IEEE Dual Logo International Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEC/IEEE Dual Logo International Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

NOTE – Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

IEEE Standard Codes, Formats, Protocols, and Common Commands for use With IEEE Std 448.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation

Sponsor
Automated Instrumentation Committee
of the
IEEE Instrumentation and Measurement Society

Approved June 18, 1992
IEEE Standards Board

Approved January 4, 1993
American National Standards Institute

Abstract: A set of codes and formats to be used by devices connected via the IEEE 488.1 bus is specified. This standard also defines communication protocols that are necessary to effect application-independent and device-dependent message exchanges, and further defines common commands and characteristics useful in instrument system applications. It is intended to apply to small-scale to medium-scale instrument systems comprised mainly of measurement, stimulus, and interconnect devices outside the scope of the instrument system environment. IEEE 488.1 subsets, standard message-handling protocols including error handling, unambiguous program and response-message syntactic structures, common commands useful in a wide range of instrument system applications, standard status reporting structures, and system configuration and synchronization protocols are covered.
Keywords: controller, device, system, system bus, system interface

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

IEEE Foreword

IEEE Std 488 has been in existence for seventeen years, during which time its power and versatility have been proven decisively. Probably its most important contribution to test system automation has been the standardization of system interconnection and communication. A large and growing base of design and application experience has allowed this further standardization of the IEEE Std 488 bus. The additional standardization sought by the original IEEE Std 488.2-1987 was founded on the premise that the existing investment in IEEE Std 488 resources must be protected. Thus, IEEE Std 488.2 describes functionality that complements and is based upon IEEE Std 488.1-1987. Every effort has been made to reap the benefits of standardization without limiting the freedom and creativity of the device designer.

Since the introduction of IEEE Std 488-1978, individual manufacturers of devices have developed internal standards that address the code, format, protocol, syntax, and semantic concepts. System integrators have identified requirements useful in configuring systems. All these sources have been used and refined in developing this standard.

This revision of the original 1987 document contains many minor changes, too numerous to list, plus a few major changes. The changes are intended to be optional additions and clarifications to the 1987 standard. The goal of the working group was to allow devices and controllers designed in compliance with IEEE Std 488.2-1978 to remain in compliance with IEEE Std 488.2-1992.

Major new features include the addition of an <EXPRESSION RESPONSE DATA> element; the expanded use of <SUFFIX PROGRAM DATA>; the addition of the *RMC, Remove Individual Macro Command; the addition of the *SDS, Save Default Device Settings Command; the modification of *DDT, *DMC, *PUD, and *RDT commands to accept block or string data; and a new appendix on Reset Guidelines. There are many clarifications including a macro expansion algorithm; the removal of ambiguity in the definition of suffixes; and the scope of *LRN?, *SAV, *RCL, and *RST, to name a few.

The hope is that this new revision, incorporating five years of experience with IEEE Std 488.2-1987, will provide an even more solid foundation for the standardized communication of test and measurement systems.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

STANDARD DIGITAL INTERFACE FOR PROGRAMMABLE INSTRUMENTATION -

Part 2: Codes, formats, protocols and common commands

1. Introduction

1.1 Scope

This standard specifies a set of codes and formats to be used by **devices** connected via the IEEE 488.1 bus. This standard also defines communication protocols necessary to effect application independent device-dependent message exchanges and further defines common commands and characteristics useful in instrument **system** applications.

This standard is intended to apply directly to small-scale to medium-scale instrument **systems**. It applies to **systems** comprised mainly of measurement, stimulus, and interconnect **devices** with an instrumentation **controller**. The standard may also apply to certain **devices** outside the scope of the instrument **system** environment.

As well as defining a variety of device-dependent messages, this standard extends and further interprets certain interface functions contained in IEEE Std 488.1-1987 [4][1] while remaining compatible with that standard.

This standard covers the following topics:

1) IEEE 488.1 subsets
2) Standard message handling protocols including error handling
3) Unambiguous program and response message syntactic structures
4) Common commands useful in a wide range of instrument **system** applications
5) Standard status reporting structures
6) **System** configuration and synchronization protocols

Use of this standard does not relieve the user from the burden of responsibility for **system** compatibility at the application level. The user must be familiar with the characteristics of all the **system** components in order to configure an optimum **system**.

The intended readers of this standard include both **controller** and **device** designers.

[1]The numbers in brackets correspond to those of the references in Section 2.

--`,,```,,,-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

1.2 Objectives

The objectives of this standard are

1) To provide a well-defined and unambiguous structure of codes, formats, protocols, and common commands
2) To retain generality to accommodate the needs of a wide variety of applications within the scope of the standard
3) To promote the degree to which **devices** from different manufacturers may be interconnected and used, without modification
4) To enable the interconnection of instrumentation and related **devices** with both limited and extensive capability to generate, process, and interpret a variety of different message types
5) To define codes, formats, protocols, and common commands that will reduce the costs of generating application software and the costs of **system** integration
6) To permit direct communication among instrument **system devices** without extraordinary translation and conversion of special codes and formats

1.3 Notation

This standard defines several common English words that have special meaning in the context of this standard. These words have different connotations in IEEE Std 488.1-1987 [4], but are, for reasons of readability and conciseness of text, the preferred words for use in this standard. To avoid confusion, these words always appear in bold type when referenced in this document. Words that appear in bold type are: **system, device, controller, system bus,** and **system interface**. Local messages peculiar to IEEE 488.2 will also appear in lowercase bold type.

This standard also defines syntactic elements that are used to describe messages transferred on the bus. Syntactic elements are enclosed by angle brackets, for example, <syntactic element>, to set them off from local messages, remote messages, and normal text.

2. References

This standard shall be used in conjunction with the following publications:

[1] ANSI X3.4-1986, American National Standard Code for Information Interchange Coded Character Set—7-Bit.[2]

[2] ANSI X3.42-1990, American National Standard Representation of Numeric Values in Character Strings for Information Interchange.

[3] IEEE Std 260-1978, (Reaff 1985). IEEE Standard Letter Symbols for Units of Measurement (SI Units, Customary Inch-Pound Units, and Certain Other Units) (ANSI, DoD).[3]

[4] IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation (ANSI)*.

[2]ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.
[3]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

* Reference is made in this standard to IEEE Std 488.1:1987, an earlier edition of the IEEE document upon which IEC/IEEE 60488-1:2004 is based.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

[5] IEEE Std 754-1985 (Reaff 1991), IEEE Standard for Binary Floating-Point Arithmetic.

[6] ISO 31-1: 1992, Quantities and units — Part 1: Space and time.[4]
[7] ISO 31-2: 1992, Quanities and units — Part 2: Periodic and related phenomena.

[8] ISO 31-3: 1992, Quantities and units — Part 3: Mechanics.

[9] ISO 31-4: 1992, Quanitites and units — Part 4: Heat.

[10] ISO 31-5: 1992, Quanitites and units — Part 5: Electricity and magnetism.

[11] ISO 31-6: 1992, Quantities and units — Part 6: Light and related electromagnetic radiations.

[12] ISO 31-7: 1992, Quantities and units — Part 7: Acoustics.

[13] ISO 31-8 : 1992, Quantities and units — Part 8: Physical chemistry and molecular physics.

[14] ISO 31-9: 1992, Quantities and units — Part 9: Atomic and nuclear physics.

[15] ISO 31-10: 1992, Quantities and units — Part 10: Nuclear reactions and ionizing radiations.

[16] ISO 31-11: 1992, Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology.

[17] ISO 31-12: 1992, Quantities and units — Part 12: Characteristic numbers.

[18] ISO 1000: 1992, SI units and recommendations for the use of their multiples and of certain other units.

[19] ISO 2955: 1983, Information processing—Representation of SI and other units in systems with limited character sets.

[20] Mueller, J. E., "Efficient Instrument Design Using IEEE 488.2," *IEEE Transactions on Instrumentation and Measurement*, vol. IM-39, pp. 146–150, Feb. 1990.

3. System Considerations

3.1 Definitions

The following definitions apply for the purpose of this standard. This section contains only general definitions. Detailed definitions are given in further sections, as appropriate. For definitions relating specifically to IEEE Std 488.1-1987 [4], see 1.3 of that standard.

controller: . The component of a **system**that functions as the **system controller**. A **controller**typically sends program messages to and receives response messages from **devices**. A controller may pass and receive control per the protocols in this standard. A controller meets all the requirements stated in Section 14. of this standard.

default: . The choice used when no specification is given.

[4]ISO publications are available from ISO, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

device: A component of a **system** that does not function as the **system controller** but typically receives program messages from and sends response messages to the **controller**. A **device** may optionally have the capability to receive control from the **controller** and become the controller-in-charge of the **system.** A **device**meets all the requirements stated in Section 4. of this standard.

system: . A group of **devices** and a **controller** interconnected with a **system interface**.

system bus: The IEEE 488.1 bus and protocols that interconnect the **devices** and **controllers** in a **system.** The content of this standard applies to device-dependent traffic over this bus.

system interface: An interface that connects a **device** or **controller** to the **system bus**. A "non-IEEE 488.2 system interface" is any interface other than the **system interface** that may happen to be connected to a **device** or **controller**

3.2 System Message Traffic

This standard is optimized for a **system** in which the **devices** do not become controller-in-charge. The usual message traffic is assumed to be from "controller-to-device" or from "device-to-controller" (see Fig 3-1).

The flexibility of the program message syntax, see Section 7., also may allow the use of an IEEE 488.2 component in a **system** that contains non-IEEE 488.2 components.

In a **system** with a **device** and a noncompliant **controller**, the flexible listening formats allow the achievement of a higher degree of compatibility. This compatibility is realized by requiring minor variations in syntax to be accepted by the **device.** These variations are designed to be syntactically easy to generate on a variety of **controllers.** They may even be the noncompliant **controller's** default syntax. Such **systems** are beyond the scope of this standard and may not perform their intended functions.

3.2.1 Program and Response Message Relationships

Subsequent sections of this standard will describe, in detail, the allowed syntax and semantics for **system** message traffic. The philosophy of this standard is that **devices** receive in a more flexible manner than they send.

Thus, a range of variations of syntax defined by this standard must be accepted when **devices** listen. Likewise, a precise syntax also identified by this standard is required when **devices** talk.

These controller-to-device (program) and device-to-controller (response) messages are composed of syntactic elements that are described in Sections 7. and 8., respectively.

The following example demonstrates this relationship among program and response messages for a **device** that has a range that may be programmed and queried. The **device** has discrete ranges of 1.2, 12, and 120.

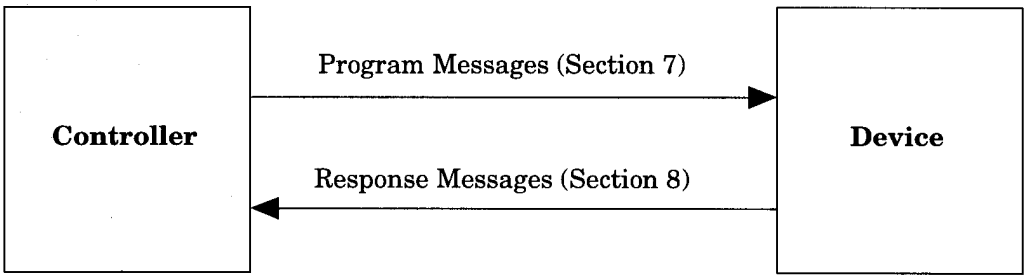


Figure 3-1—Usual Message Traffic

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The following is an example of flexible reception of range programming with the **device** listening.

Example of Flexible Reception

Controller Sends	Device Interprets
RANGE 12.45	RANGE 12
or RANGE 12	RANGE 12
or RANGE 1.2E+1	RANGE 12

The syntactic rules of Section 7. provide for this flexibility of interpretation.

The following is an example of precise response to range status request with the **device** talking.

Example of Precise Response **Controller**

Controller Sends	Device Sends
RANGE?	12

The syntactic rules of Section 8. ensure that the **device** responds with a precise format.

3.2.2 Other Message Traffic

In addition to controller-to-device and device-to-controller messages, protocols are defined to facilitate passing control, see Section 17. Device-to-device protocols are not explicitly defined in this document. However, message traffic between **devices** shall follow the syntax for <RESPONSE MESSAGES> defined in Section 8.

3.3 Functional Layers

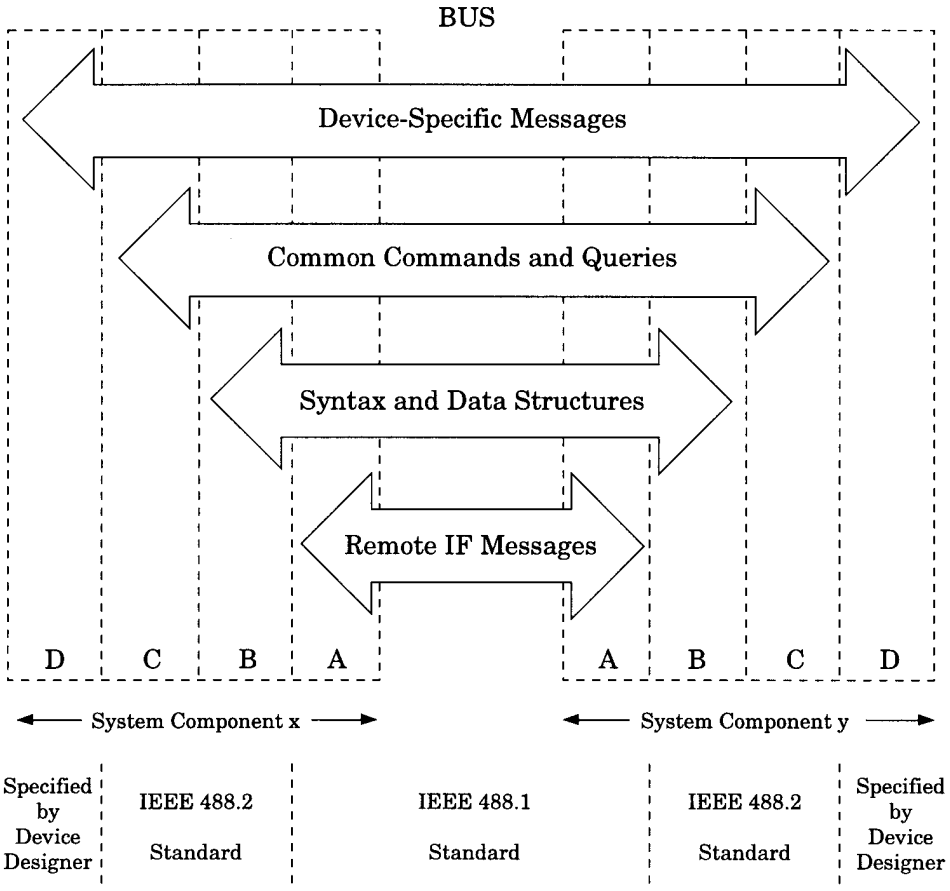
The purpose of an interface **system** is to support the **system** application of the user. In the schema of which this standard is a part, the **system** contains successive, independent "levels" of communication protocol, as shown in Fig 3-2.

Each layer shown has associated common "messages" that form protocols to communicate between itself and the corresponding layer in the participating **system** components.

The protocols are designed for communication between nonpeer entities due to the requirements of IEEE 488.1-1987 [4]. No attempt has been made to associate the protocol layering with the ISO Model for Open Systems Interconnection (OSI).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



The layers have the following representations:

Layer D represents Device Functions
Layer C represents Common System Functions
Layer B represents Message Communication Functions
Layer A represents Interface Functions (IF)

Layers D, C, and B contain IEEE 488.1 device-dependent messages and layer A contains IEEE 488.1 interface messages.

Figure 3-2—IEEE 488.1 and IEEE 488.2 Functional Protocol Layers

4. Device Compliance Criteria

A **device** shall have certain capabilities. This section lists the capabilities that this standard requires in a **device.** A **device** may optionally contain additional capabilities. Any optional **device** capabilities that are described by this standard are also listed in this section.

Compliance for a **device** is divided into several areas that are considered separately. A **device** shall satisfy all the required functionality in each of the areas in order to comply with this standard. This section summarizes requirements

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

that are fully specified in the referenced section. The **device** designer shall use the requirements stated in later sections when actually designing a **device.**

4.1 IEEE 488.1 Requirements

A **device** shall contain the IEEE 488.1 subsets listed in Table 4-1 and no others.

Table 4-1—IEEE 488.1 Interface Requirement for Devices

IEEE 488.1 Interface Function	IEEE 488.1 Subsets	IEEE 488.2 Section
Source Handshake	SH1	5.1.1
Acceptor Handshake	AH1	5.1.2
Talker	T5, T6, TE5, or TE6	5.3
Listener	L3, L4, LE3, or LE4	5.4
Service Request	SR1	5.5
Remote Local	RL0 or RL1	5.6
Parallel Poll	PP0 or PP1	5.7
Device Clear	DC1	5.8
Device Trigger	DT0 or DT1	5.9
Controller	C0, or C4 with C5, C7, C9, or C11	5.10
Electrical Interface	E1 or E2	5.11

A **device** shall comply with IEEE 488.1. It shall also meet all requirements stated in Section 5. of this standard.

4.2 Message Exchange Requirements

A **device** shall follow all the requirements listed in Section 6. of this standard.

A **device's** Input Buffer may take on several forms. The Input Buffer's length may be a fixed number of bytes. It may contain a fixed number of complete <PROGRAM MESSAGE> elements. Its length in bytes or <PROGRAM MESSAGE> elements may vary with the **device's** state. See 6.1.5.

Certain query messages may generate the actual response message when the query is received. Other query messages may generate the actual response when the **controller** reads the response. See 6.4.5.4.

Execution of individual parsable elements may be done as they are received. The **device** may also wait until either a <PROGRAM MESSAGE UNIT SEPARATOR> or a <PROGRAM MESSAGE TERMINATOR> is parsed before executing preceding <PROGRAM MESSAGE UNIT> elements. A **device** may contain a mix of these types of commands. See 6.4.5.1.

4.3 Syntax Requirements

Sections 7. and 8. describe a set of functional elements that the **device** designer uses to describe the programming language for a specific **device**. The entire syntax described in Sections 7. and 8. is not required in every **device**. Some functional elements are required and some are optional.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.3.1 Required Functional Elements

Table 4-2 lists the required functional elements.

Table 4-2—Required Functional Elements for Devices

Device Listening Functional Element	Section
<PROGRAM MESSAGE>	7.3.2
<PROGRAM MESSAGE TERMINATOR>	7.5.0
<PROGRAM MESSAGE UNIT>	7.3.2
<PROGRAM MESSAGE UNIT SEPARATOR>	7.4.1
<COMMAND MESSAGE UNIT>	7.3.2
<QUERY MESSAGE UNIT>	7.3.2
<COMMAND PROGRAM HEADER>*	7.6.1
<QUERY PROGRAM HEADER>	7.6.2
<PROGRAM HEADER SEPARATOR>	7.4.3
<PROGRAM DATA SEPARATOR>	7.4.2
<PROGRAM DATA>	7.3.2
<DECIMAL NUMERIC PROGRAM DATA>	7.7.2

* NOTE — <compound command program header> and <compound query program header> are not required encoding elements.

Device Talking Functional Element	Section
<RESPONSE MESSAGE>	8.3.2
<RESPONSE MESSAGE TERMINATOR>	8.5.0
<RESPONSE MESSAGE UNIT>	8.3.2
<RESPONSE MESSAGE UNIT SEPARATOR>	8.4.1
<RESPONSE DATA>	8.7.0
<RESPONSE DATA SEPARATOR>	8.4.2
<NR1 NUMERIC RESPONSE DATA>	8.7.2
<ARBITRARY ASCII RESPONSE DATA>	8.7.11

4.3.2 Optional Functional Elements

Table 4-3 lists the optional functional elements.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-3—Optional Functional Elements for Devices

Device Listening Functional Element	Section
<COMMAND PROGRAM HEADER>*	7.6.1
<QUERY PROGRAM HEADER>	7.6.2
<CHARACTER PROGRAM DATA>	7.7.1
<SUFFIX PROGRAM DATA>	7.7.3
<NONDECIMAL NUMERIC PROGRAM DATA>	7.7.4
<STRING PROGRAM DATA>	7.7.5
<ARBITRARY BLOCK PROGRAM DATA>	7.7.6
<EXPRESSION PROGRAM DATA>	7.7.7

* NOTE — If a **device** implements <compound command program header> and <compound query program header> elements, it does not implement <simple command program header> and <simple query program header> elements. See 7.6.1 and 7.6.2.

Device Talking Functional Element	Section
<RESPONSE HEADER SEPARATOR>	8.4.3
<RESPONSE HEADER>	8.6
<CHARACTER RESPONSE DATA>	8.7.1
<NR2 NUMERIC RESPONSE DATA>	8.7.3
<NR3 NUMERIC RESPONSE DATA>	8.7.4
<HEXADECIMAL NUMERIC RESPONSE DATA>	8.7.5
<OCTAL NUMERIC RESPONSE DATA>	8.7.6
<BINARY NUMERIC RESPONSE DATA>	8.7.7
<STRING RESPONSE DATA>	8.7.8
<DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>	8.7.9
<INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>	8.7.10
<EXPRESSION RESPONSE DATA>	8.7.12

4.4 Status Reporting Requirements

4.4.1 Required Status Reporting Capability

A **device** shall follow the status reporting model presented in Section 11. The **device** shall include those commands related to status reporting described in Section 10. and listed in Table 4-4.

A **device** shall implement the status byte register, the Service Request Enable Register, the Standard Event Status Register, and the Standard Event Status Enable Register illustrated in Fig 4-1.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

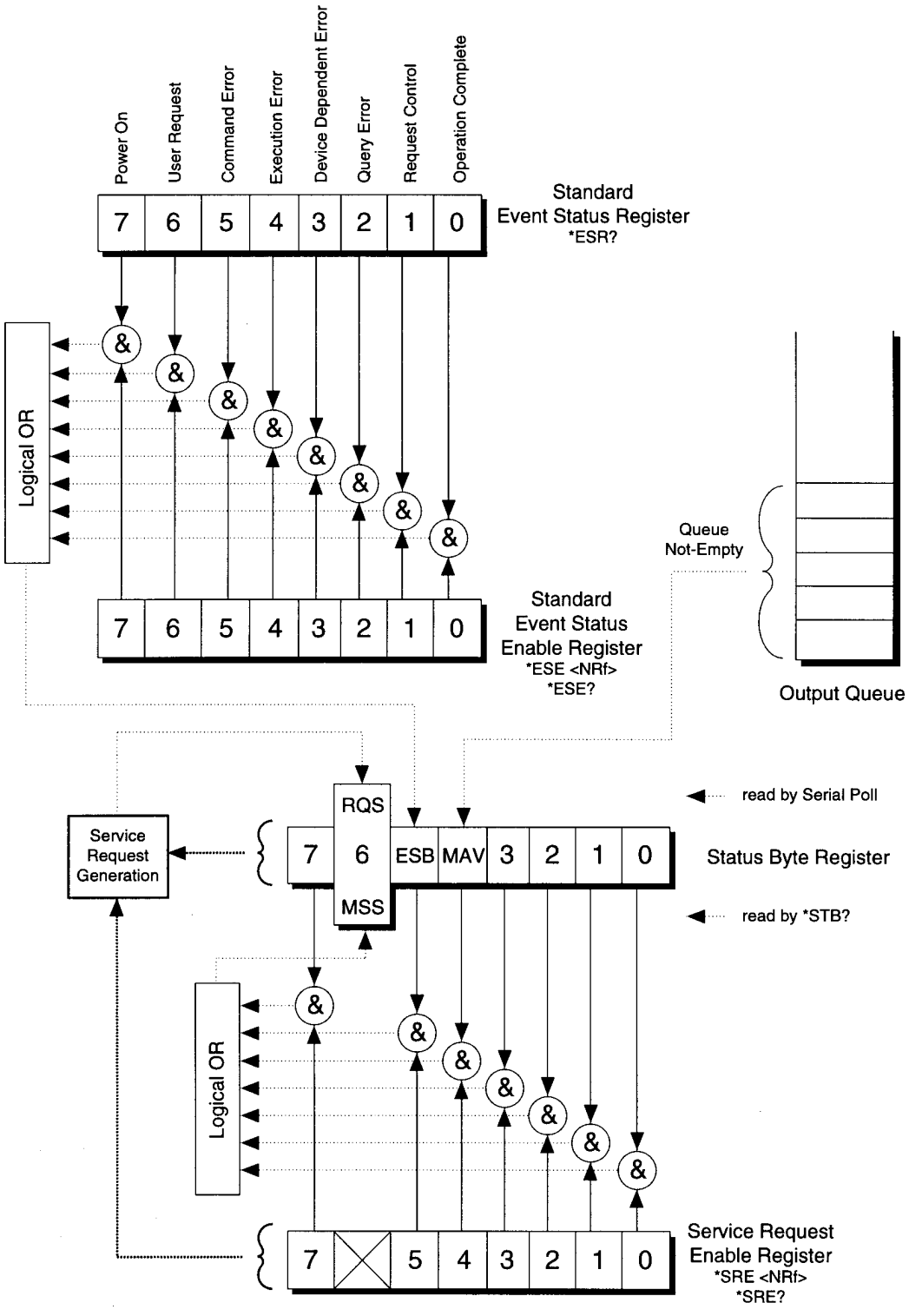


Figure 4-1—Required Status Reporting Capabilities

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-4—Required Status Reporting Common Commands

Status Reporting	
Command	**Section**
*CLS	10.3
*ESE?	10.10
*ESE?	10.11
*ESR?	10.12
*SRE?	10.34
*SRE?	10.35
*STB?	10.36
The asterisk symbol(*), preceding CLS in the above table and throughout the text, is meant to literally represent the 1st character of a common command or query (see also 7.6.1.2, 8.6.2, and Table 9-2).	

4.4.2 Optional Status Reporting Capability

A **device** may include any number of condition registers, event registers, enable registers, and queues, provided that they follow the model presented in Section 11.

The **device** may optionally include the ability to save enable registers when power is cycled. This capability also requires nonvolatile memory and all of the commands listed in Table 4-5.

Table 4-5—Optional Power-On Common Commands

Power On	
Command	**Section**
*PSC	10.25
*PSC?	10.26

The **device** may optionally include the ability to respond to a parallel poll. This capability also requires the IEEE 488.1 subset PP1 and all of the commands listed in Table 4-6.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-6—Optional Parallel Poll Common Commands

Parallel Poll	
Command	**Section**
*IST?	10.15
*PRE	10.23
*PRE?	10.24

4.5 Common Commands

This standard lists some reserved commands. Some of the commands are required, some are optional, and some shall be implemented in groups.

4.5.1 Required Common Commands

The common commands listed in 4.4.1 that are related to status reporting and the common commands listed in 4.6 that are related to synchronization are required. The commands that relate to internal operations are listed in Table 4-7 and are required.

Table 4-7—Required Internal Operation Common Commands

Internal Operations	
Command	**Section**
*IDN?	10.14
*RST	10.32
*TST?	10.38

4.5.2 Optional Common Commands

In some cases, the implementation of a common command is independent of other common commands or **device** capabilities. In other cases, common commands shall be implemented in groups or in conjunction with some other **device** capability.

4.5.2.1 Resource Description Commands

The resource description commands, listed in Table 4-8, are optional and loosely coupled. If the resource description can be written into the **device** (*RDT), then the **device** shall also include the capability to read the resource description (*RDT?). The *RDT? query, however, may be included without the *RDT command.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-8—Optional Resource Description Common Commands

Resource Description	
Command	**Section**
*RDT	10.30
*RDT?	10.31

4.5.2.2 Protected User Data Commands

The protected user data commands, listed in Table 4-9, are optional. If either command is implemented, then both commands shall be implemented.

Table 4-9—Optional Protected User Data Commands

Protected Data	
Command	**Section**
*PUD	10.27
*PUD?	10.28

4.5.2.3 Calibration Command

The self-calibration command, listed in Table 4-10, is optional.

Table 4-10—Optional Calibration Command

Calibration	
Command	**Section**
*CAL?	10.2

4.5.2.4 Trigger Command

The trigger command, listed in Table 4-11, is optional. It is required, however, if the **device** has DT1 capability. If the trigger command is implemented, the **device** shall also have DT1 capability.

Table 4-11—Optional Trigger Command

Trigger	
Command	**Section**
*TRG	10.37

4.5.2.5 Trigger Macro Commands

The trigger macro commands, listed in Table 4-12, are optional. Implementing either command requires the **device** to have DT1 capability. Implementing *DDT requires the **device** to implement *DDT?.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-12—Optional Trigger Macro Commands

Trigger Macro	
Command	**Section**
*DDT	10.4
*DDT?	10.5

If the trigger macro commands are implemented, the **device** shall also include the <ARBITRARY BLOCK PROGRAM DATA> and <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> functional elements.

4.5.2.6 Macro Commands

The macro commands, listed in Table 4-13, are optional. If any commands in this group are implemented, however, then all commands in this group shall be implemented.

If the macro commands are implemented, the **device** shall also include the <STRING PROGRAM DATA>, <ARBITRARY BLOCK PROGRAM DATA>, <STRING RESPONSE DATA>, and <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> functional elements.

4.5.2.7 Option Identification command

The option identification command, listed in Table 4-14, is optional.

Table 4-13—Optional Macro Commands

Macros	
Command	**Section**
*DMC	10.7
*EMC	10.8
*EMC?	10.9
*GMC?	10.13
*LMC?	10.16
*PMC	10.22

Table 4-14—Optional Option Identification Command

Options	
Command	**Section**
*OPT?	10.20

4.5.2.8 Stored Setting Commands

The stored setting commands, listed in Table 4-15, are optional. If either command is implemented, then both commands shall be implemented.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-15—Optional Stored Setting Commands

Stored Settings	
Command	**Section**
*RCL	10.29
*SAV	10.33

4.5.2.9 Learn Command

The learn command, listed in Table 4-16, is optional.

Table 4-16—Optional Learn Command

Learn	
Command	**Section**
*LRN?	10.17

4.6 Synchronization Requirements

Section 12. describes the capability that is required in all **devices** for synchronization. A **device** designer may choose which operations have pending-operation flags associated with them in accordance with the rules in Section 12.

The required common commands related to synchronization are listed in Table 4-17.

Table 4-17—Required Synchronization Commands

Synchronization	
Command	**Section**
*OPC	10.18
*OPC?	10.19
*WAI	10.39

4.7 System Configuration Capability

Section 13. describes the optional capability that is used to configure the IEEE 488.1 addresses of **devices**. This capability is implemented by including the common commands in Table 4-18.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-18—Optional System Configuration Commands

Auto Configure	
Command	**Section**
*AAD	10.1
*DLF	10.6

If either command is implemented, then both commands shall be implemented.

These commands cause the **device** to suspend temporarily normal message exchange protocols and to use different protocols. To ensure compatibility with other **devices**, the **device** designer must carefully adhere to these different protocols.

4.8 Controller Capability

4.8.1 Required Controller Capability

A **device** is not required to have any **controller** capability.

4.8.2 Optional Controller Capability

If a **device** has any **controller** capability, it shall include the IEEE 488.1 C4 subset and the pass control back command listed in Table 4-19. It shall obey the passing control protocol described in 17.4 and it shall obey the requesting control protocol described in 17.5.

Table 4-19—Optional Passing Control Command

Stored Settings	
Command	**Section**
*PCB	10.21

4.9 Device Documentation Requirements

The documentation provided with a **device** shall describe how the **device** has implemented this standard. This documentation is required and shall include

1) A list of IEEE 488.1 Interface Functions subsets implemented, see Section 5.
2) A description of **device** behavior when the address is set outside the range of 0–30, see 5.2.
3) A description of when a user-initiated address change is recognized by the **device**.
4) A description of the **device** setting at power-on, see 5.12. Any commands that modify the power-on settings shall also be included. A list of device-specific enable registers affected by the *PSC common command shall be included.
5) A description of message exchange options:
   a) The size and behavior of the Input Buffer, see 6.1.5.
   b) Which queries return more than one <RESPONSE MESSAGE UNIT>, see 6.4.3.
   c) Which queries generate a response when parsed, see 6.4.5.4.
   d) Which queries generate a response when read, see 6.4.5.4.
   e) Which commands are coupled, see 6.4.5.3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6) A list of functional elements used in constructing the device-specific commands. Whether <compound command program header> elements are used must also be included, see 7.1.1 and 7.3.3.
7) A description of any buffer size limitations related to block data, see 7.7.6.5.
8) A list of the <PROGRAM DATA> elements that may appear within an <expression> as well as the maximum subexpression nesting depth. Any additional syntax restrictions that the **device** may place on the <expression> shall also be included.
9) A description of the response syntax for every query, see Section 8.
10) A description of any device-to-device message transfer traffic that does not follow the rules for <RESPONSE MESSAGE> elements, see 8.1.
11) The size of any block data responses, see 8.7.9.4.
12) A list of common commands and queries that are implemented, including, where appropriate, indications of any optional functional elements implemented, see Section 10..
13) A description of the state of the **device** after successful completion of the Calibration query, see 10.2.
14) The maximum length of the block used to define the trigger macro (see 10.4.6.1) and the method of interpreting *TRG within a *DDT command sequence (see 10.4.6.3).
15) The maximum length and complexity of macro labels; the maximum length of the block used to define a macro; and how recursion is handled during macro expansion, if the macro commands are implemented, see 10.7.
16) A description of the response to the identification common query, *IDN?, see 10.14.
17) If the *DDT command is implemented,
    a) The maximum length of the block used to define the trigger macro, see 10.4.
    b) The command sequence sent with the *DDT command has the same effect as *RST on the action performed by GET, see 10.4.1 and 10.4.6.3.
18) The size of the resource description, if the *RDT command or *RDT? query are implemented, see 10.30 and 10.31.
19) A description of the states affected by *RST (see 10.32), *LRN? (see 10.17), *RCL (see 10.29), and *SAV (see 10.33).
20) A description of the scope of the self-test performed by the *TST? query, see 10.38.
21) A description of additional status data structures used in the **device's** status reporting, see Section 11.
22) For each command, a statement describing whether it is overlapped or sequential.
23) For each command, the **device** documentation shall specify the functional criteria that are met when an operation complete message is generated in response to that command, see 12.8.3.
24) A description of the representations, if any, used for infinity and not-a-number, see 7.7.2.4.5 and 8.7.4.4.

5. Device Interface Function Requirements

This section describes the IEEE 488.1 interface function requirements of a **device.** It specifies the additional requirements of a **device** that are directly associated with the ten IEEE 488.1 interface functions. These requirements are intended to supplement the IEEE 488.1 specification for the IEEE 488.2 **system** environment as described in this standard.

5.1 Handshake Requirements

5.1.1 Source Handshake Requirements

A **device** shall contain the SH1 (complete capability) subset. It shall make the transition from Source Delay State (SDYS) to Source Transfer State (STRS) only if the ready for data (RFD) message is TRUE and the data accepted (DAC) message is FALSE, see 6.5.3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

5.1.2 Acceptor Handshake Requirements

A **device** shall contain the AH1 (complete capability) subset and shall additionally follow the requirement of the following paragraph.

A **device** shall enter Acceptor Idle State (AIDS) of the AH function within 1 ms after ATN assumes the FALSE state if the **device** was in Listener Idle State (LIDS) when ATN was TRUE. This requirement ensures that the **device** will operate reliably with the Find Listeners common **controller** protocol, see 17.3.

5.2 Address Requirements

A **device** shall have the same talk and listen addresses. The lower 5 bits of the MTA and MLA codes, the primary address of the **device,** shall be identical. (See IEEE Std 488.1-1987 [4], 6.3.) If the **device** utilizes extended addressing, the talk and listen secondary addresses (MSAs) shall also be identical.

A **device** shall have a single primary address that may be set by the user to any value within the range of 0 to 30. If the **device** utilizes extended addressing, the user shall also be able to set the single secondary address within the same range.

The address of a **device** shall be settable independent from any other **device's** address even if the **devices** share common physical resources such as the same enclosure.

A **device** shall have a local means of selecting its IEEE 488.1 primary address (and secondary address if extended addressing is used) that shall be retained during power-off. The **device** operator shall be capable of altering this address.

The physical configuration, labeling, and positioning of the address selection mechanism shall follow the guidelines of IEEE Std 488.1-1987 [4], Appendix I.

The **device** designer should avoid using address settings outside the range of 0-30. Any exceptions to this recommendation shall be documented with the associated behavior.

If hard switches are used to set the address, and the address is apparently set to 31, the **device** shall operate in a manner that does not disrupt **system bus** traffic.

Device documentation shall state when a user-initiated address change is recognized by the **device.** A **device** shall update its address at power-on.

5.3 Talker Requirements

A **device** shall contain either the T5, T6, TE5, or TE6 subsets. These subsets require the basic talker with serial poll and unaddress if MLA.

This standard assumes that the **controller** performs all talker addressing via MTA and unaddressing via MLA, OTA, and UNT. Talk-only mode capability is, therefore, not a requirement of this specification. The presence of a talk-only mode of operation (for example, in a **controller**-less **system**) is allowable, but operation in this mode is considered beyond the scope of this standard.

5.4 Listener Requirements

A **device** shall contain either the L3, L4, LE3, or LE4 subsets. These subsets require the basic listener with unaddress if MTA.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

This standard assumes the **controller** performs all listener addressing via MLA and unaddressing via MTA and UNL. Listen-only mode capability is, therefore, not a requirement of this specification. The presence of a listen-only mode of operation (for example, in a **controller**-less **system**) is allowable, but operation in this mode is considered beyond the scope of this standard.

Note the additional requirement for inactivating the Acceptor Handshake function on an ATN FALSE transition in Listener Idle State (LIDS). See 5.1.2.

5.5 Service Request Requirements

A **device** shall contain the SR1 (complete capability) subset and shall conform to the status handling requirements of Section 11.

5.6 Remote/Local Requirements

5.6.1 Control and Operation Definitions

IEEE 488.2 control functions and operations are classified using the source of control and method of annunciation.

5.6.1.1 IEEE 488.2 Remote Operation

An IEEE 488.2 remote operation is any operation of a **device** function in a **system** that is effected via program messages over the **system interface**.

5.6.1.2 IEEE 488.2 Local Operation

An IEEE 488.2 local operation is any operation of a **device** function that is not an IEEE 488.2 remote operation.

An example is the actuation of a user-accessible switch, knob, button, touch-screen location, etc., that is physically attached to a **device** and used locally to control or program the **device.**

Operation of a **device** via any other bus or interface connected to the **device** is considered to be local operation from the standpoint of the **system.**

5.6.1.3 Local Control

A local control effects IEEE 488.2 local operation of a **device.**

Local controls include **device** inputs that are designed for control.

Local controls and the accompanying Remote/Local requirements of this section are intended to apply only to local controls whose function can also be effected via IEEE 488.2 remote operation.

Local controls whose function cannot be effected via IEEE 488.2 remote operation or whose function is to protect the safety of the user or equipment are beyond the scope of this standard.

The following control functions are explicitly excluded from categorization as local controls.

1) Switching line power.
2) The generation of the user request (URQ) message.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

5.6.1.4 External-Control-Signal

An external-control-signal invokes **device** actions using a **device** port other than the **system interface.** External-control-signals are beyond the scope of this standard, but may optionally function as local controls. External-control-signals, however, shall not violate the message exchange protocols of Section 6.

An example of an external-control-signal is a digital voltmeter's external trigger, which initiates a measurement. The results may then be read from the **device.**

5.6.1.5 Hard Local Control

Hard local controls on a **device** have indicators (mechanical, positional, etc.) that cannot be changed by IEEE-488.2 remote operation of the **device.**

For example, an instrument may have a mechanical rotary switch that is used to select among three different modes of operation. The state of this switch is indicated by printed labels on the front panel of the **device,** but remote messages cannot rotate the switch.

5.6.1.6 Soft Local Control

Any local control that is not a hard local control is a soft local control. For example, a physical (momentary contact) key on a **device** may be used to turn a **device** function on or off with alternate pushes. The on/off state is indicated by a light located by the switch. The light is controlled by the actual state of the **device** function.

5.6.1.7 Programmable Local Control

A local control of a **device** function that can be affected by IEEE 488.2 remote operation as well as by IEEE 488.2 local operation is a programmable local control. Programmable local controls may be hard or soft local controls.

5.6.2 IEEE 488.1 Subset Requirements

A **device** that is capable of IEEE 488.2 local and remote operation and that utilizes programmable local controls shall contain the RL1 (complete capability including local lockout) subset of the IEEE 488.1 Remote/Local function.

A **device** that is incapable of IEEE 488.2 local operation or that does not utilize programmable local controls may contain the RL0 (no capability) subset of the IEEE 488.1 Remote/Local function. A **device** containing the RL0 subset shall ignore all IEEE 488.1 interface messages related to Remote/Local function state changes.

5.6.3 Local-to-Remote State Transition Requirements

The transition from Local State (LOCS) or Local With Lockout State (LWLS) to Remote State (REMS) or Remote With Lockout State (RWLS) shall disable local operation of all programmable local controls.

5.6.4 Remote-to-Local State Transition Requirements

The transition from REMS or RWLS to LOCS or LWLS shall enable local operation of all programmable local controls of the **device.**

The transition shall also alter any hard local control functions so that they match the hard local control indicator. After the transition, the **device's** front panel and state of control shall agree.

The transition shall not alter any soft local control functions.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Setting the rtl local message TRUE with LLO false or Accept Data State (ACDS) inactive shall cause a **device** in REMS to change to LOCS (see IEEE Std 488.1-1987 [4], 2.8.3.3).

5.6.5 Local State Operation

In LOCS or LWLS, a **device** shall process all program messages so as to maintain consistency between the information presented to the local user and the actual state of the **device.**

To avoid an inconsistency, a **device** shall do one of the following:

1) Correct the inconsistent information presented to the local user
2) Remove the inconsistent information
3) Generate an execution error and discard the responsible <PROGRAM MESSAGE UNIT> (See IEEE 488.2 status reporting protocols in Section 11.)

Alternative (1) is the preferred choice.

In LOCS or LWLS, a **device** shall not be inhibited from sending a <RESPONSE MESSAGE> in response to a query previously received by the **device.**

When a **device** is sent program messages while in local, there are potential arbitration problems if front panel controls are manipulated. Application programs can avoid such problems by placing the **device** in remote.

5.6.6 Remote State Operation

In REMS or RWLS, a **device** shall process all program messages.

In REMS or RWLS, device-specific messages (not defined by this standard) may be sent to selectively enable specific local controls. Under this circumstance, the specified local controls are no longer locally inoperative. If specific hard local controls are enabled, the **device's** state of control and these hard local controls shall agree.

In REMS or RWLS, a **device** shall not be inhibited from sending a <RESPONSE MESSAGE> in response to a query previously received by the **device.**

5.6.7 Operation Independent of Remote/Local State

The Remote/Local function shall be associated only with the enabling and disabling of local controls as defined in this section. Other **device** operation, such as updating the **device** state and associated local annunciation, shall function independent of the Remote/Local state of the **device.**

5.6.8 Remote/Local Indicator Requirements

Devices with hard local controls shall incorporate a “Remote” indicator (see IEEE Std 488.1-1987 [4], Appendix H).

For **devices** without hard local controls, the Remote indicator is optional. All indicators relating to Remote/Local transitions shall be labeled and shall function in accordance with the requirements of IEEE Std 488.1-1987 [4], Appendix H.

5.7 Parallel Poll Requirements

A **device** shall have either the PP0 (no capability) or the PP1 (remote configuration) subset. A **device** with the PP1 subset shall conform to the requirements of 11.6.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

5.8 Device Clear Requirements

A **device** shall contain the DC1 (complete capability including selective **device** clear) subset. Specific IEEE 488.2 requirements for the **device's** response to entering DCAS are specified in Section 6. and summarized in the following:

A **device** entering Device Clear Active State (DCAS) via a DCL or SDC command shall:

1) Clear the Input Buffer and Output Queue.
2) Reset the Parser, Execution Control, and Response Formatter (see 6.1.4.2.6).
3) Clear any command that prevents processing a *RST or other **device** commands.
4) Discard all commands and queries deferred due to coupled parameters (see 6.4.5.3).
5) Put the **device** into Operation Complete Command Idle State (OCIS) and into Operation Complete Query Idle State (OQIS) (see 12.5.2.1.1 and 12.5.3.1.1).
6) Abort the *AAD and *DLF common commands (see Section 13.).
7) Put the **device** in the message exchange IDLE state (see 6.3.1.1).

Shall not:

1) Change any settings or stored data in the **device** except as stated earlier.
2) Interrupt front panel I/O.
3) Interrupt or affect any **device** operation in progress except as stated earlier.
4) Change the status byte other than clearing the MAV bit as a consequence of clearing the Output Queue, (see 11.2.1.2).

See Appendix D for a discussion of the resetting actions of **dcas** as they relate to other reset commands.

5.9 Device Trigger Requirements

A **device** may contain either the DT0 (no capability) or the DT1 (complete capability) IEEE 488.1 **device** trigger function. A **device** containing the DT1 subset shall implement the associated common command, *TRG, as defined in 10.37. Use of the optional *DDT common command, see 10.4, affects the operation of *TRG and Group Execute Trigger (GET).

5.10 Controller Function Requirements

A **device** shall contain either no Controller function capability (CO) or shall contain a Controller function with the following subset options: C4 with C5, C7, C9, or Cll. These subsets provide the capability to respond to SRQ, send IF messages, receive control, pass control, and take control synchronously. They specifically prohibit **system controller** capability, that is, sending the IFC and REN interface messages. They optionally permit the capability of passing control to self and conducting a parallel poll.

If a **device** contains any Controller function subset other than CO, then it shall also be able to pass and receive control by means of the protocols described in 17.4 and 17.5.

See Section 15. for **controller** (as contrasted to **device**) requirements.

5.11 Electrical Requirements

Implementation of the E2 electrical interface option (see IEEE Std 488.1-1987 {4}, Appendix C2) is recommended. Open collector drivers shall be used to drive the SRQ, NRFD, and NDAC signal lines. Three-state drivers should be used to drive the DAV, EOI, and ATN signal lines. A **device** is not allowed to drive REN or IFC. If the **device** is in Parallel Poll Active State (PPAS), the **device** shall use open collector drivers to drive the DIO1-8 signal lines. When not

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

in PPAS, the **device** should use three-state drivers to drive the DIO1-8 signal lines. If a **device** needs to source handshake data bytes at a rate greater than 250 000 bytes per second, the E2 option is required (see IEEE Std 488.1-1987 {4}, 5.2).

NOTE — Some integrated circuits may require external hardware to avoid driving the DIO lines during a parallel poll, even though the PP0 subset is implemented.

5.12 Power-On Requirements

At power-on, **device** settings shall be either restored to their values when the **device** was last powered off, set to known states that are explicitly stated in the **device** documentation, or set to a state defined by the user and stored in local nonvolatile memory.

Device-specific commands may be provided to select whether **device** settings are restored or set to a known state at power-on. This standard defines one such command, power-on status clear (*PSC, see 10.25.4), which sets and clears the power-on-status-clear flag.

See Appendix D for a discussion of the resetting actions of power-on and *PSC as they relate to other reset commands.

5.12.1 Items Not Affected by Power-On

A **device** shall not alter the following due solely to a power-on (receipt of the IEEE 488.1 pon local message):

1) The bus address
2) Pertinent calibration data
3) Data or **device** states that cause a change in the response to the following common queries:
   a) *IDN? (Identification Query, see 10.14)
   b) *OPT? (Option Identification Query, see 10.20)
   c) *PSC? (Power-On Status Clear Query, see 10.26)
   d) *PUD? (Protected User Data Query, see 10.28)
   e) *RDT? (Resource Description Transfer Query, see 10.31)

5.12.2 Items Dependent Upon Power-On-Status-Clear Flag

If the power-on-status-clear flag is FALSE, the Service Request Enable Register, the Standard Event Status Enable Register, and the Parallel Poll Enable Register shall not be affected by power-on. See *PSC command in 10.25.

A **device** shall clear the Service Request Enable Register, the Standard Event Status Enable Register, the Parallel Poll Enable Register, and device-specific event enable registers at power-on if the power-on-status-clear flag is TRUE or if the *PSC command is not implemented.

The **device** designer may use the state of the power-on-status-clear flag to control whether other status registers are cleared at power-on.

5.12.3 Items That May be Affected by Power-On

At the discretion of the **device** designer, a **device** may alter the following at power-on:

1) Current **device** function states
2) Status information, see 5.12.2 and 11
3) *SAV/*RCL registers, see 10.33 and 10.29
4) Macro definition defined with the *DDT command, see 10.4
5) Macro definitions defined with the *DMC command, see 10.7

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

6) Macro enabling with the *EMC command, see 10.8
7) Address received with last *PCB command, see 10.21

6. Message Exchange Control Protocol

The **device** message exchange protocol specifies how a **device** handles program and response messages.

The term **controller** refers to the **system** element that is exchanging messages with the **device**. The requirements of this section also apply to device-to-device communication.

The phrase "... the **device** shall (not) ..." defines a requirement of this standard. The phrase "... the **controller** should (not) ..." refers to conditions that will (not) happen under normal conditions. The **device's** response to both normal and exceptional conditions is defined by this standard.

The term "Query Message," as used in this section, is a <PROGRAM MESSAGE> that contains one or more <QUERY MESSAGE UNIT> elements. Note that all bracketed syntactic elements are defined in Sections 7. and 8.

The terms "Command Error," "Execution Error," "Query Error," and "Standard Event Status Register" are defined in Section 11.

6.1 Functional Elements

Fig 6-1 presents the relationship between the IEEE 488.1 bus, the Message Exchange Interface described in this section, the Status Reporting functions described in Section 11., and the device-specific functions provided by the **device** designer.

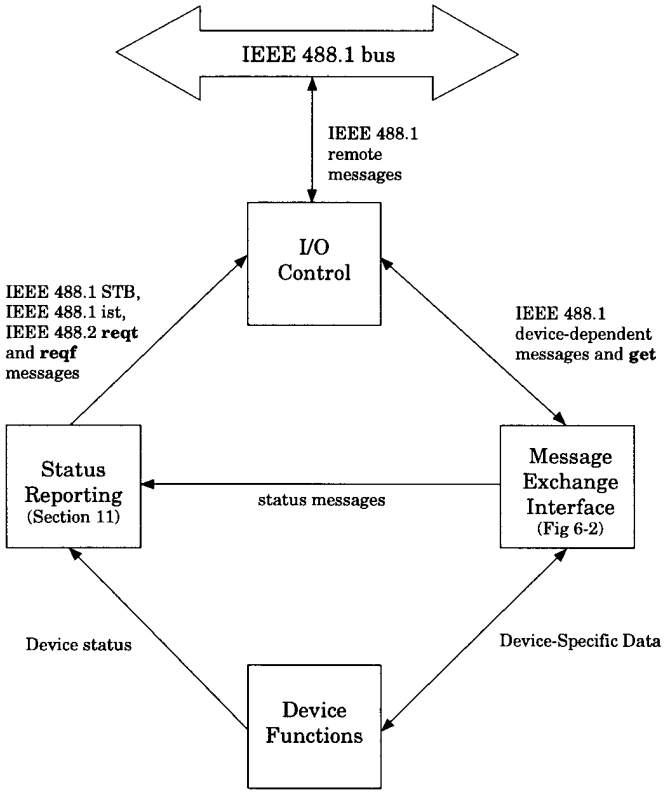


Figure 6-1—Device Status and Message Exchange Overview

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.1 IEEE 488.1 Bus

The IEEE 488.1 bus, shown in both Figs 6-1 and 6-2, represents the physical IEEE 488.1 interface cable and drivers as defined in IEEE Std 488.1-1987 [4].

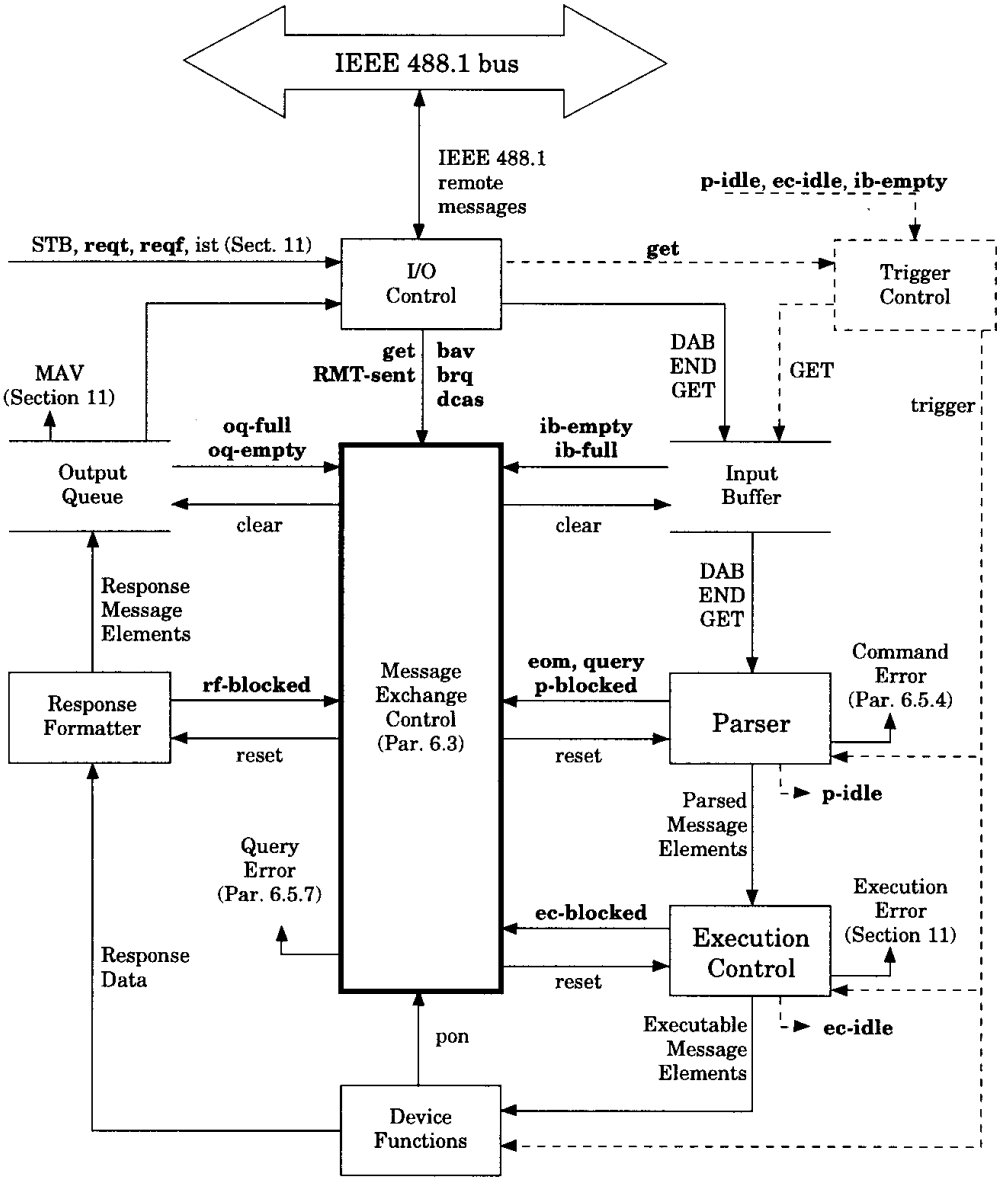


Figure 6-2—Message Exchange Control Interface Functional Blocks

6.1.2 Status Reporting

The Status Reporting block appears only in Fig 6-1. Fig 6-2 does, however, display the status messages sent from the Message Exchange Interface and the messages sent to the I/O Control.

The Status Reporting block receives error and status messages from the Message Exchange Interface and Device Functions blocks. It also sends the STB, **reqt, reqf,** and IEEE 488.1 ist messages to the I/O Control, so that it can

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

respond to EEE 488.1 Serial Poll and Parallel Poll requests. The Status Reporting block is described completely in Section 11.

6.1.3 Message Exchange Interface

The Message Exchange Interface deals with all IEEE 488.1 device-dependent messages to and from the **device.**

Fig 6-2 presents a logical model that describes the operation of the **device's** Message Exchange Interface. It is not intended to imply a particular physical implementation. Some handshaking signals between blocks have been omitted to reduce the complexity of the diagram.

NOTE — For an example of an implementation of the Message Exchange Interface, see [20].

6.1.4 I/O Control

The I/O Control, shown in Figs 6-1 and 6-2, interprets and decodes IEEE 488.1 remote messages. Many of the functions of this block are provided by commercial integrated circuits that have been designed to implement the requirements of IEEE 488.1.

The I/O Control responds to IEEE 488.1 Serial Poll and Parallel Poll requests as directed by the STB, **reqt, reqf,** and ist messages from the Status Reporting block, see Section 11. It generates the **dcas,** DAB, END, **get, bav,** and **brq** messages that are sent to the Message Exchange Interface, and receives the DAB and END messages from the Message Exchange Interface.

6.1.4.1 I/O Control Rules

6.1.4.1.1 I/O Control Sending Data Bytes

When **brq** is TRUE and **oq-empty** is FALSE, the I/O Control shall remove one byte from the Output Queue, set **brq** FALSE, and send the byte to the **controller** using the IEEE 488.1 Source Handshake (SH) function. See 6.1.4.2.3.

6.1.4.1.2 I/O Control Receiving Data Bytes

When **bav** is TRUE and **ib-full** is FALSE, the I/O Control shall set **bav** FALSE and then place the associated DAB and any accompanying END message into the Input Buffer using the IEEE 488.1 Acceptor Handshake (AH) function. See 6.1.4.2.4.

6.1.4.1.3 I/O Control Receiving IEEE 488.1 GET Message

When **get** is TRUE and ib-full is FALSE, the I/O Control shall set **get** FALSE and then place a GET message in the Input Buffer.

NOTE — If the optional Trigger Control is implemented, the *get* message shall be sent to the Trigger Control rather than the Input buffer. The *get* can be executed directly under certain conditions, see 6.1.11.

6.1.4.1.4 I/O Control Ordering of Received Messages

The I/O control shall preserve the order of received messages, including data bytes (DABs), END messages, Group Execute Trigger, Device Clear, and Selected Device Clear.

The following example shows a design technique for preserving the order of received data bytes and the IEEE 488.1 remote interface messages: DCL, SDC, and GET.

Design the **device's** message interface for Device Clear handling as follows:

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

1) Design the IEEE 488.1 AH function to wait in ACDS when the Device Clear (DC) function enters the Device Clear Active State (DCAS) until the I/O Control sets the **dcas** message TRUE.
2) Ensure that the I/O Control sets **bav** FALSE and discards the DAB before it releases the bus holdoff and allows the **device's** AH function to accept a new data byte. Since the new data byte associated with the **bav** message must have been received first with the bus being in a holdoff state during the receipt of the DCL or SDC messages, proper sequencing is guaranteed.

Design the **device's** message interface for Device Trigger handling as follows:

1) Design the IEEE 488.1 AH function to wait in Accept Data State (ACDS) when the Device Trigger (DT) function enters the Device Trigger Active State (DTAS) after a GET is received and until the I/O Control sets the **get** message TRUE. This action is often called "NDAC bus holdoff."
2) Have the I/O Control set **bav** FALSE and place the data byte in the Input Buffer before accepting the GET message when the **bav** and **get** messages are both TRUE. Since the new data byte associated with the **bav** message must have been received first with the bus being in a holdoff state during the receipt of the GET message, proper sequencing is guaranteed.
3) Ensure that the I/O Control accepts the GET message (sets get TRUE) before it releases the bus holdoff and allows the **device's** IEEE 488.1 AH function to accept a new data byte.

NOTE — The use of NDAC holdoff on GET allows the following situation when the Input Buffer is full. The **device** will assert NDAC as long as the Input Buffer is full. The **controller** cannot communicate with the **device.** A Device Clear cannot be sent. The only way to remove this condition is to have

— The parser remove a DAB, END, or GET from the Input Buffer, or

— The Device Function send the pon message to the Message Exchange Control

6.1.4.2 I/O Control Messages

This section describes the action of the I/O control IEEE 488.2 local messages.

6.1.4.2.1 Data Byte Message (DAB)

The DAB message is defined in IEEE 488.1. It consists of eight bits that are either sent or received by the **device** while ATN is FALSE.

6.1.4.2.2 End Message (END)

The END message is defined in IEEE 488.1. It is sent with a DAB by setting EOI TRUE and ATN FALSE.

6.1.4.2.3 Byte Requested Message (brq)

The **brq** message indicates the readiness of the I/O Control to send a data byte to the **controller.**

The I/O Control shall set **brq** TRUE after the IEEE 488.1 SH function enters the IEEE 488.1 Source Generate state (SGNS) and the T or TE function is in the Talker Active State (TACS). The **brq** message shall be set TRUE once and only once for each transition into SGNS.

The I/O Control shall set **brq** FALSE when any of the following conditions occur:

1) The I/O Control sends a data byte from the Output Queue to the **controller** (see 6.1.4.1.1).
2) The **device** performs the INTERRUPTED action described in 6.3.2.3.
3) The **device** performs the UNTERMINATED action described in 6.3.2.2.
4) The I/O Control sets the **dcas** message TRUE.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

5) The Message Exchange Control enters the IDLE state from the DONE or DEADLOCK states. No other **device** or interface condition shall cause **brq** to be set FALSE.

6.1.4.2.4 Byte Available Message (bav)

The **bav** message indicates that the I/O Control has received a data byte and is ready to place it in the Input Buffer.

The I/O Control shall set **bav** TRUE after the IEEE 488.1 L or LE function is in the Listener Active State (LACS) and the IEEE 488.1 Acceptor Handshake function enters the Accept Data State (ACDS). The **bav** message shall be set TRUE once and only once for each transition from the Acceptor Ready State (ACRS) to ACDS.

The I/O Control shall set **bav** FALSE when any of the following conditions occur:

1) A data byte is placed in the Input Buffer. (If the **device** has a zero length Input Buffer, the I/O Control shall set **bav** FALSE when the data byte is sent to the Parser.) See 6.1.4.1.2.
2) The **dcas** message is set TRUE. The associated data byte shall be discarded.

No other **device** or interface condition shall cause **bav** to be set FALSE. See 6.1.4.1.3.

6.1.4.2.5 Group Execute Trigger Message (get)

The IEEE 488.2 **get** message signals the transition of the IEEE 488.1 Device Trigger (DT) function to the Device Trigger Active State (DTAS). This transition occurs when a IEEE 488.1 Group Execute Trigger (GET) remote interface message is received while the **device** was addressed to listen. The **get** message can be used to initiate a device-defined action or the action defined by the *DDT common command, see 10.4.

The **device** shall process data bytes and GET messages in the order received, see 6.1.4.1.4.

This standard specifies two methods for **device** triggering, the common command *TRG, see 10.37, and the IEEE 488.1 interface message GET. Typically, **device** triggering is used for two distinct purposes:

1) For the properly sequenced initiation of a device-defined or user-defined action within a single addressed **device**
2) For the synchronized initiation of device-defined or user-defined actions within multiple addressed **devices**

For purpose (1), either GET or *TRG may be used for **device** triggering. The storage of either one in the Input Buffer ensures that the trigger will be executed in the correct sequence in relation to other commands.

For purpose (2), where the required synchronization could be compromised by the software/ firmware processing overhead, the optional Trigger Control Block may be used. When the Trigger Control Block receive a **get** message from the I/O Control Block, it generates a **trigger** local message for immediate execution by the **device,** provided that the Input Buffer is empty and the Parser and Execution Control are idle.

If the Input Buffer is not empty or if the Parser or Execution Control are not idle, the alternative GET message shall be placed in the Input Buffer, and execution of the trigger shall be deferred until previous commands have been executed.

If the optional Trigger Control block (see 6.1.11) is implemented, the I/O Control shall send the **get** message to it. Otherwise, the I/O Control shall place the GET message directly in the Input Buffer.

6.1.4.2.6 Device Clear Active State Message (dcas)

The I/O Control shall set the **dcas** message TRUE after the **device** enters the Device Clear Active State (DCAS) defined by IEEE 488.1 . The *device* enters DCAS whenever it receives the Device Clear (DCL) remote command or it receives the Selected Device Clear (SDC) remote command while it is addressed to listen.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

When **dcas** is set TRUE, the Input Buffer and Output Queue shall be cleared; the Parser, Execution Control, and Response Formatter shall be reset; and the **device** shall enter the Message Exchange IDLE state. The **device** also enters Operation Complete Command Idle State (OCIS) and Operation Complete Query Idle State (OQIS), see 12.5, and returns to normal operation after the *AAD or *DLF commands, see Section 13.

The I/O Control shall set the **dcas** message FALSE when these actions are completed and the Message Exchange Control enters the IDLE state. Device Clear requirements are summarized in 5.8.

The **device** shall ensure that data bytes and DCL or SDC are processed in the order received. If the *device* receives a new data byte while **dcas** is TRUE, it shall not discard the data byte or set **bav** FALSE.

6.1.4.2.7 Response Message Terminator Sent Message (RMT-sent)

The I/O Control block shall set the **RMT-sent** message TRUE when it has sent the <RESPONSE MESSAGE TERMINATOR>.

Receipt of TRUE **bav** or **brq** messages shall cause the I/O Control to set the **RMT-sent** message FALSE.

6.1.5 Input Buffer

The Input Buffer, shown in Fig 6-2, stores DABs, GET messages, and END messages. The Input Buffer then delivers these messages to the Parser in the order that they were received from the I/0 Control. The **dcas** message shall not be stored in the Input Buffer, as this could result in the **device** not being able to immediately respond to **dcas.**

The Input Buffer shall be implemented as a first-in first-out data structure. Data bytes, END, and GET messages are placed into the Input Buffer by the I/O Control as the **device** handshakes them from the IEEE 488.1 bus. The details of when and how items are placed into the Input Buffer are described in 6.1.4.1.2, 6.1.4.1.3, and 6.1.4.1.4 on I/O Control Receiving. DABs, END, and GET messages are removed from the Input Buffer by the Parser at a rate consistent with the capabilities of the Parser and the associated execution of commands.

6.1.5.1 Input Buffer Rules

6.1.5.1.1 Input Buffer Length

The Input Buffer shall be equal to or greater than zero in length. The Input Buffer's length may be fixed at some number of bytes or program messages, or its length may vary. The Input Buffer length shall be specified in the user documentation.

The operation of the Input Buffer is transparent to application programs, except as a performance improvement. **Devices** that can accept program messages faster than they can execute them should have Input Buffers long enough to allow the storing of a reasonably sized program message. This buffering will allow application programs to send messages to the **device** and then continue to use the **system bus** for other purposes while the **device** is taking the time it needs to respond.

For example, a digital plotter responds to commands slowly due to the need to allow for pen movement. If a typical length program message for the **device** is 80 bytes, the **device** should have an Input Buffer of at least 80 bytes.

The **device** designer must consider the application of the **device** when choosing an Input Buffer length. A larger Input Buffer is useful in smart, highly-independent test instruments that use complex <PROGRAM MESSAGE> elements. These complex <PROGRAM MESSAGE> elements may require a significant time to process. In some **systems,** this time can be used for tasks other than waiting for this **device** to finish processing a <PROGRAM MESSAGE>. Increasing buffer length, however, requires the user to be more careful about synchronization within the **system.**

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

If a zero length Input Buffer is used, the **system interface** provides a source of synchronization. With this very short Input Buffer, the **device** can never fall behind the **controller**. If a DAB or GET is sent, it will be processed immediately by the **device**.

NOTE — Most commercial integrated circuits that implement the requirements of IEEE 488.1 provide at least a 1 byte internal data buffer.

6.1.5.1.2 Input Buffer Overflow

The Input Buffer shall not overflow. If the buffer is full, the **device** shall not enter the IEEE 488.1 Acceptor Ready State (ACRS) while in Listener Active State (LACS) (NRFD holdoff). After the Parser removes an item from the Input Buffer, the I/O control may accept another item from the **system interface**. No error shall be reported when the Input Buffer fills unless the **device** becomes deadlocked, see 6.3.1.7 and 6.5.7.4.

6.1.5.1.3 Input Buffer Clearing

The Input Buffer shall be cleared when pon or **dcas** is TRUE. No **device** or interface condition is allowed to keep the clear operation from taking place. Except for the emptying during normal message processing and the actions of pon and dcas, no other interface or **device** condition shall be allowed to clear the Input Buffer. The Input Buffer is cleared to ensure that the **device** will be ready to receive and execute new <PROGRAM MESSAGE> elements following **dcas.**

6.1.5.2 Input Buffer Messages

6.1.5.2.1 Input Buffer Empty Message (ib-empty)

The Input Buffer Empty message, **ib-empty**, shall be TRUE when the parser attempts to remove an item from the Input Buffer and the Input Buffer is empty. The ib-empty message shall be set FALSE when I/O Control Block places a DAB, END, or GET message into the Input Buffer.

6.1.5.2.2 Input Buffer Full Message (ib-full)

The Input Buffer Full message, ib-full, shall be TRUE when the Input Buffer is full. The **ib-full** message shall be FALSE at all other times.

6.1.6 Parser

The Parser, shown in Fig 6-2, is the logical portion of a **device** that takes DABs, END messages, and GET messages from the Input Buffer and analyzes them by separating out the various IEEE 488.2 syntactic elements. It reports invalid syntax or headers to the Status Reporting block as Command Errors. The Parser converts syntactic elements into an internal representation which is sent to the Execution Control. The Parser also generates the **eom** and **query** messages when it recognizes these syntactic elements.

A <PROGRAM MESSAGE> or <PROGRAM MESSAGE UNIT> is considered “parsed” when it has been analyzed by the Parser and the Parser is ready to continue analyzing other <PROGRAM MESSAGE UNIT> elements.

6.1.6.1 Parser Rules

6.1.6.1.1 Parser Errors

The Parser shall detect Command Errors and report them to the Status Reporting block by setting the Command Error bit in the Standard Event Status Register, see 11.5.1.1.4.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The Parser shall report a Command Error if it finds a syntax error in a <PROGRAM MESSAGE>, if it finds an unrecognized header, if it finds a parameter (<PROGRAM DATA> element) that is of the wrong type for its associated header, or if it encounters a GET message between the first byte of a <PROGRAM MESSAGE> and the <PROGRAM MESSAGE TERMINATOR>.

When a Command Error is detected, the **device** shall determine what will be done with any prior parsable elements of the same <PROGRAM MESSAGE>. The **device** is allowed to either discard or attempt to execute any such parsable elements.

When a Command Error occurs, the Parser shall discard all subsequent DABs and GET messages until either

1) **dcas** is TRUE.
2) pon is TRUE.
3) **eom** is TRUE.
4) **brq** is TRUE and **ib-empty** is TRUE.

The **device** designer may also choose to stop discarding DABs and GET messages for other device-defined conditions such as receipt of a comma, semicolon, or NL. The Parser is reset when any of these conditions is met.

NOTE — The **device** does not empty the Input Buffer after a Command Error. The Parser discards DABs and GET messages until it has detected one of the above conditions. The **device** will then resume normal parsing and executing of messages.

6.1.6.1.2 Parser Resetting

When the Parser is reset, it shall interpret the next data byte it receives as the first byte of a <PROGRAM MESSAGE>. The Parser shall be reset when **dcas** or pon is TRUE. No **device** or interface condition is allowed to keep the reset operation from taking place. Other conditions that can reset the Parser are described in 6.1.6.1.1.

6.1.6.2 Parser Messages

6.1.6.2.1 Parsed Message Element

A Parsed Message Element represents the **device's** internal representation of all or part of a <PROGRAM MESSAGE>.

6.1.6.2.2 End of Message Detected Message (eom)

The Parser shall set the End of Message Detected message, **eom**, TRUE when any of the following conditions occurs:

1) When it receives an END message or a sequence of data bytes making up a <PROGRAM MESSAGE TERMINATOR>, see 7.5, from the Input Buffer
2) After it parses a GET message, except when the GET occurs between the first byte of a <PROGRAM MESSAGE> and the <PROGRAM MESSAGE TERMINATOR>, see 6.1.6.1.1.
3) When the Trigger Control block has passed the **trigger** message to the Device Functions block

The Parser shall set the **eom** message FALSE when any of the following conditions occurs:

1) When it is reset
2) When it receives any other DAB or GET message from the Input Buffer

When the Parser has received a GET message from the Input Buffer or the Trigger Control block has passed a **trigger** message to the Device Functions block, any associated **query** message shall be set TRUE prior to setting the **eom** message TRUE. This ensures a proper sequence of execution of Message Exchange block transitions, see Fig 6-4.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

6.1.6.2.3 Query Detected Message (query)

The Parser shall set the Query Detected message, **query,** TRUE when it receives any of the following:

1) A sequence of data bytes making up a valid <QUERY PROGRAM HEADER> that is not a macro label
2) A valid <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> that is a macro label, where the macro is defined to include a <QUERY MESSAGE UNIT>
3) A GET message or *TRG command (or, if implemented, the **trigger message**), where the **device** implements the *DDT common command and the **device** trigger action defined by the *DDT command includes a <QUERY MESSAGE UNIT>
4) A GET message or *TRG command (or, if implemented, the **trigger message**), where the **device** does not implement the *DDT common command and the **device** trigger action specified by the **device** designer is to generate a <RESPONSE MESSAGE>

"Valid <QUERY PROGRAM HEADER>" means that all characters up to and including the "?" represent a valid query mnemonic or macro label, and the characters immediately following the "?" form a legal separator or terminator.

"Valid <COMMAND PROGRAM HEADER>" means that the header represents a valid command mnemonic or macro label, and the characters immediately following the mnemonic form a legal separator or terminator.

For example,

*XYZ? is not a valid <QUERY PROGRAM HEADER> because there is no "*XYZ?" common query.

*ESR?X is not a valid <QUERY PROGRAM HEADER> because "X" is not a separator or terminator.

*IDN? 42 is a valid <QUERY PROGRAM HEADER>, although the argument is incorrect. A Command Error will be generated for all of the three conditions listed above.

The Parser shall set **query** FALSE when it is reset or when it receives any other DAB or GET message from the Input Buffer.

6.1.6.2.4 Parser Idle Message (p-idle)

The Parser shall set the Parser Idle message, **p-idle**, TRUE when the Parser has parsed all prior messages and is not actively processing a valid <COMMAND PROGRAM MESSAGE> or a valid <QUERY PROGRAM MESSAGE>. The Parser shall set **p-idle** FALSE at all other times.

6.1.6.2.5 Parser Blocked message (p-blocked)

The Parser shall set the Parser Blocked message, **p-blocked,** TRUE when the Parser is waiting for the Execution Control to finish processing the previous. Parsed Message Element. The Parser shall set **p-blocked** FALSE at all other times.

6.1.7 Execution Control

The Execution Control block, shown in Fig 6-2, deals with coupled parameters (see 6.4.5.3), queries that require a **device** action before they can generate a <RESPONSE MESSAGE>, and the **device** synchronization commands described in Section 12.. The Execution Control determines when the **device** has enough information to execute a command. When the Execution Control has gathered enough information to initiate a **device** action, it sends an Executable Message Element to the Device Functions block. The Execution Control may wait for the resulting **device** actions to complete .(Sequential Command) or may continue processing commands (Overlapped Command), depending on the command. See 12.2. A <PROGRAM MESSAGE UNIT> is considered "executed" when it has been

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

parsed and all corresponding **device** operations have been either completed (Sequential Command) or initiated (Overlapped Command).

6.1.7.1 Execution Control Rules

6.1.7.1.1 Execution Control Errors

The Execution Control shall detect Execution Errors and report them to the Status Reporting block, see 11.5.1.1.5.

6.1.7.1.2 Execution Control Resetting

When the Execution Control is reset, prior history will not affect the execution of new <PROGRAM MESSAGE> elements received after dcas or pen. The Execution Control shall be reset when INTERRUPTED, or when **dcas** or pen is TRUE. No **device** or interface condition is allowed to keep the reset operation from taking place. No other interface or **device** condition shall reset the Execution Control.

6.1.7.2 Execution Control Messages

6.1.7.2.1 Executable Message Element

An Executable Message Element is a directive to the Device Functions block to perform some device-specific action. This may take the form of a Valid Query Request or a nonquery Executable Message Element.

A Valid Query Request is an Executable Message Element that causes the Device Functions block to send data to the Response Formatter. A nonquery element performs device-specific action but does not send Response Data to the Response Formatter. A single <PROGRAM MESSAGE UNIT> may cause more than one Executable Message Element to be sent.

For example, a query that starts a measurement and returns the results when the measurement is complete might require two Executable Message Elements: a nonquery element to start the measurement and a Valid Query Request to send the results to the Response Formatter.

The Execution Control also shall maintain the order of creating <RESPONSE MESSAGE UNIT> elements, as required by 6.4.5.4, by sending the Valid Query Requests that are associated with previous received <QUERY MESSAGE UNITS> that are already sent to the Device Functions block.

6.1.7.2.2 Execution Control Idle Message (ec-idle)

The Execution Control shall set the Execution Control Idle Message, **ec-idle,** TRUE when the following conditions occur:

1) All deferred commands have been executed.
2) Execution Control is remedy to accept a new command or query from the Parser, and
   a) Execution Control is reset, or
   b) All previous commands have been executed

The Execution Control shall set **ec-idle** FALSE at all other times.

6.1.7.2.3 Execution Control Blocked Message (ec-blocked)

The Execution Control shall set the Execution Control Blocked message, **ec-blocked,** TRUE when it is ready to send a Valid Query Request to the Device Functions block but must wait for the Response Formatter and Device Functions block to process a prior Valid Query Request. The Execution Control shall set **ec-blocked** FALSE at all other times.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

6.1.8 Device Functions

The Device Functions block, shown in both Figs 6-1 and 6-2, contains all device-specific features and functions. It accepts Executable Message Elements from the Execution Control and performs the associated operations.

6.1.8.1 Device Functions Rules

The Device Functions block shall accept Valid Query Requests from the Execution Control and shall send any required Response Data to the Response Formatter. All implemented IEEE 488.2 common query commands and all device-specific query commands shall be handled in this manner.

The Device Functions block shall report **device** status information to the Status Reporting block, Section 11.

6.1.8.2 Device Functions Messages

6.1.8.2.1 Power-On Message (pon)

The Device Functions block shall generate the pon local message as described in IEEE 488.1.

When pon is set TRUE, the Input Buffer and Output Queue shall be cleared; the Parser, Execution Control, and Response Formatter shall be reset; and the **device** shall enter the Message Exchange IDLE state. The **device** also enters Operation Complete Command Idle State (OCIS) and Operation Complete Query Idle State (OQIS), see 12.5. The pon message may also affect device-specific settings, see 5.12. The Device Functions block shall set the pon message FALSE when these actions are completed and the Message Exchange Control enters the IDLE state, see 6.3.1.1.

6.1.8.2.2 Response Data

Response Data is data provided by the **device** in response to <QUERY MESSAGE UNIT> elements that have been parsed and sent to the **device** in the form of Executable Message Elements (Valid Query Requests). Response Data is unformatted and may only be tokens representing the actual data to be formatted.

6.1.9 Response Formatter

The Response Formatter, shown in Fig 6-2, builds a <RESPONSE MESSAGE> out of Response Message Elements from Valid Query Requests and response data. The <RESPONSE MESSAGE> is placed into the Output Queue. The Response Formatter's primary responsibility is to convert the internal representation of data elements into a sequence of data bytes according to the syntax rules in Section 8.

6.1.9.1 Response Formatter Rules

The Response Formatter shall delimit all but the last <RESPONSE MESSAGE UNIT> in the <RESPONSE MESSAGE> with a <RESPONSE MESSAGE UNIT SEPARATOR> (";", see 8.4.1), and the last <RESPONSE MESSAGE UNIT> in the <RESPONSE MESSAGE> with the <RESPONSE MESSAGE TERMINATOR>.

The Response Formatter shall be reset when **dcas** or pon is TRUE. No **device** or interface condition is allowed to keep the reset operation from taking place. The Response Formatter is reset so that any new <RESPONSE MESSAGE> elements will not be affected by conditions that existed before **dcas** or pon.

6.1.9.2 Response Formatter Messages

6.1.9.2.1 Response Message Element

A Response Message Element represents a **device's** internal representation of all or part of a <RESPONSE MESSAGE>.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.9.2.2 Response Formatter Blocked Message (rf-blocked)

The Response Formatter shall set the Response Formatter Blocked message, **rf-blocked,** TRUE when it has a <RESPONSE MESSAGE UNIT> to be placed in the Output Queue and the Output Queue Full message, **oq-full**, is TRUE. The Response Formatter shall set **rf-blocked** FALSE at all other times.

6.1.10 Output Queue

The Output Queue, shown in Fig 6-2, stores device-to-controller messages until the **controller** reads them. The Response Formatter places DAB and END messages into the Output Queue in response to query commands. These bytes are removed from the Output Queue as they are read by the **controller.**

The Output Queue shown in Fig 6-2 is a logical model. The Output Queue in a real **device** may store the individual bytes of the <RESPONSE MESSAGE> or tokens that represent <RESPONSE MESSAGE> elements. The **device** designer may also use other means of determining the response to be generated.

The **device** designer should provide an Output Queue large enough to handle reasonably long <RESPONSE MESSAGE> elements. The actual length chosen will depend on the characteristics of the **device.** To avoid overflowing a physical buffer, the **device** designer may defer the formatting of lengthy <RESPONSE MESSAGE> elements until the **controller** requests output. The **device** designer should ensure that the size and operation of the Output Queue are adequate to make a deadlock unlikely, see 6.5.7.4.

NOTE — Certain integrated circuits contain a one byte buffer, which operates as part of the Output Queue when MAV is generated (see 6.1.10.2) and "Clear the Output Queue" is executed. A **device** designer cannot disregard this buffer. See 6.3.2.1(2), 6.3.2.2(2), and 6.3.2.3(2).

6.1.10.1 Output Queue Rules

The Output Queue shall be cleared when pon or **dcas** is TRUE. Note that this is not a Query Error. A Query Error shall be reported if the contents of the Output Queue are discarded for any other reason, see 6.3.1.7, 6.3.2.2, and 6.3.2.3.

6.1.10.2 Output Queue Messages

6.1.10.2.1 Message Available Message (MAV)

The Output Queue shall send the MAV message to the Status Reporting block. As long as the Output Queue contains one or more bytes, MAV shall be TRUE. MAV shall be FALSE when the Output Queue is empty, except as indicated below. See 11.2.1.2.

When the **device** defers generation of Response Data until **brq** is TRUE, the MAV message shall be set TRUE when the **device** is ready to generate the Response Data. See 6.4.5.4.

6.1.10.2.2 Output Queue Full Message (oq-full)

The Output Queue Full message, **oq-full,** shall be TRUE when the Output Queue is full. The **oq-full** message shall be FALSE at all other times.

6.1.10.2.3 Output Queue Empty Message (oq-empty)

The Output Queue Empty message, **oq-empty,** shall be TRUE when the Output Queue is empty. The **oq-empty** message shall be FALSE at all other times.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.11 Trigger Control

The Trigger Control block, shown in Fig 6-2, is optional. It allows for high-speed hardware execution of GET messages without violating the rules of execution order.

The GET message is commonly used to trigger a device-specific action or set of actions within a single addressed **device**. It is also used to trigger synchronized actions across multiple addressed **devices**. In this and other cases in which it may be necessary to avoid the software/firmware execution time overhead incurred in the Input Buffer, Parser, and Execution Control blocks, the **device** designer may choose to implement the Trigger Control block.

If the Trigger Control block is implemented, GET messages are not sent directly to the Input Buffer from the I/O Control. Instead, **get** is sent to the Trigger Control block, which either sends a **trigger** message directly to the Device Functions block, or places a GET in the Input Buffer.

6.1.11.1 Trigger Control Rules

The Trigger Control block shall detect the **get** message and check **ib-empty** (from the Input Buffer), p-idle (from the Parser), and **ec-idle** (from the Execution Control). If all four of these messages are TRUE, the Trigger Control block shall send the trigger message to the Device Functions block. If either **ib-empty, p-idle,** or **ec-idle** is FALSE, the Trigger Control block shall place the GET message in the Input Buffer.

NOTE — The implementation of this optional Trigger Control block does not affect the **device's** response to GET nor does it relax the requirement of sequential processing of GET messages and DABs. This block only facilitates highspeed operation.

6.1.12 Message Exchange Control

The Message Exchange Control represents the interconnection of control messages between the Output Queue, Response Formatter, Input Buffer, Parser, Execution Control, I/O Control, and Device Functions blocks.

6.2 Protocol Overview

This protocol overview describes the normal operation of the **device** message exchange protocol. It is not a detailed specification of the protocol, nor does it cover any protocol exceptions. The detailed specification of the **device** message exchange protocol begins in 6.3.

Fig 6-3 shows the Message Exchange states and transitions encountered during normal operation of the **device**. It does not show certain required states and transitions associated with error recovery. Paragraph 6.13 and Fig 6-4 describe the complete Message Exchange State Diagram, with the dashed lines indicating the exception or error cases.

Figs 6-3 and 6-4 use a different format from IEEE 488.1 to reinforce that Message Exchange Control is done at a level higher than IEEE 488.1 .

6.2.1 Initialization

After power on or **dcas,** the Message Exchange Control shall wait in the IDLE state for a <PROGRAM MESSAGE> or GET message from the **controller.** The **device** shall not send a <RESPONSE MESSAGE> to the **controller** until it receives a valid Query Message from the **controller.** The **controller** will not normally attempt to read data from the **device** until it has sent a Query Message to the **device.**

6.2.2 Command Processing

When the **device** receives a <PROGRAM MESSAGE> from the **controller** while in the READ state,

1) The I/O Control places the bytes of the message into the Input Buffer.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

2) The Parser removes the bytes from the Input Buffer and identifies the syntactic components of the message according to the syntax rules in Section 7.
3) The Execution Control directs the Device Functions block to perform the actions associated with the message.

The **device** may accept the bytes of another <PROGRAM MESSAGE> and place them in the Input Buffer before it has finished processing prior <PROGRAM MESSAGE> elements.

6.2.3 Query Processing

Query Processing is handled in the QUERY, SEND, RESPONSE, and DONE states. If a <PROGRAM MESSAGE> contains one or more queries, the **device** prepares a <RESPONSE MESSAGE> for the **controller** and places it in the Output Queue. After the **controller** finishes sending the Query Message to the **device,** it will address the **device** to talk and start reading the <RESPONSE MESSAGE>. The **controller** will not normally send another <PROGRAM MESSAGE> to the **device** until it has finished reading the <RESPONSE MESSAGE>.

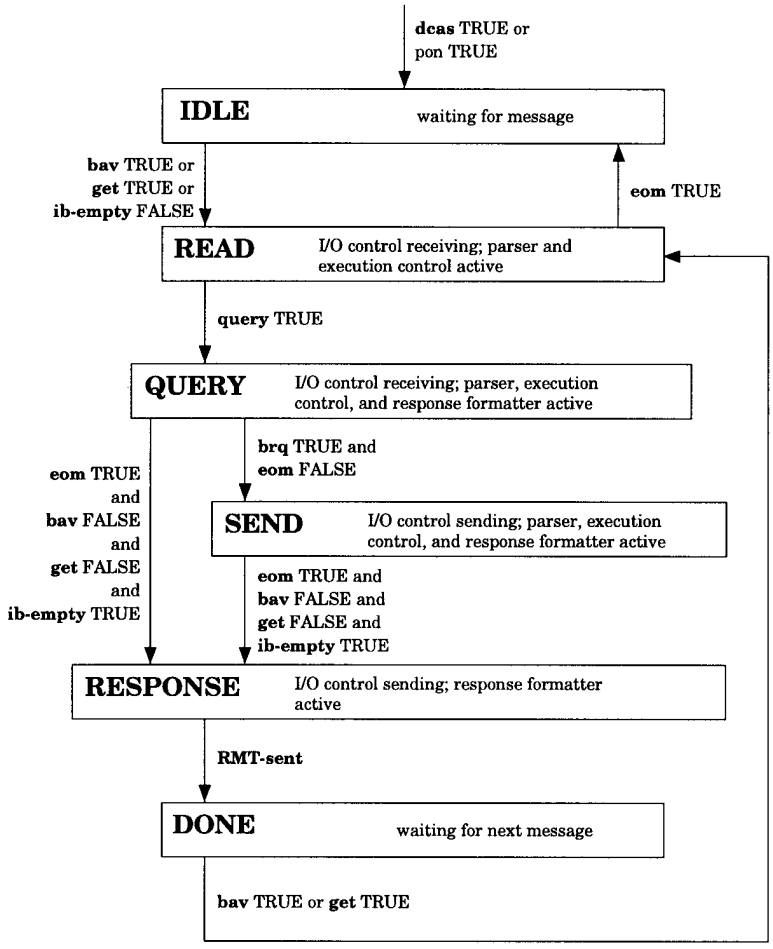


Figure 6-3—Message Exchange Control State Diagram (Simplified)

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.3 Message Exchange Control Operation

The Message Exchange Control State Diagram, Fig 6-4, precisely defines how messages from the I/0 Control, described in 6.1.4, cause the **device** to either receive or send messages. The interactions between the blocks in the Message Exchange Control Interface, Fig 6-2, are shown in detail. Fig 6-3 includes only those states and transitions encountered during "normal" operation of the **device.** Fig 6-4 includes the additional states and transitions necessary to detect Message Exchange Protocol errors. (These additional states and transitions are shown with dashed lines.)

6.3.1 Message Exchange Control States

At any time, the **device** is in one of the Message Exchange states. Depending on the state, various functional elements from Fig 6-2 are active or inactive. Transitions are caused by the logical combination of messages from the functional elements.

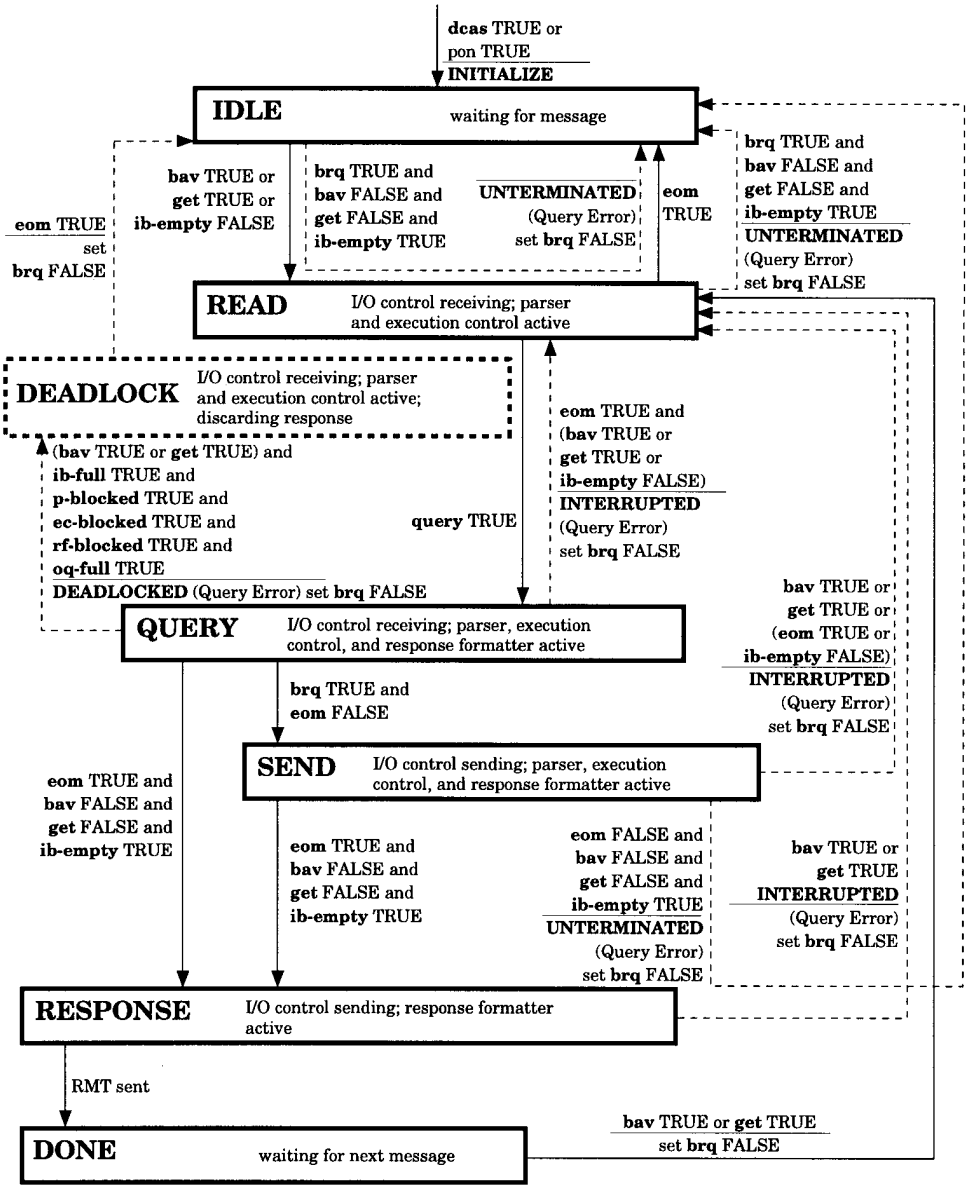


Figure 6-4—Message Exchange control State Diagram (Complete)

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.3.1.1 IDLE State

In the IDLE state, the **device** is waiting for a message from the **controller.** The next data byte received shall be interpreted as the beginning of a new <PROGRAM MESSAGE>. The Output Queue is empty.

The Message Exchange Control shall enter the READ state when either

1) The I/O Control sets **bav** TRUE, indicating that a data byte is available.
2) The I/O Control sets **get** TRUE.
3) **ib-empty** is FALSE. This condition can occur when the **device** has entered the IDLE state from the READ or DEADLOCK state.

The Message Exchange Control shall stay in the IDLE state when

1) **brq** is TRUE.
2) **bav** is FALSE.
3) **get** is FALSE.
4) **ib-empty** is TRUE.

This occurs when the **device** has been addressed to talk and has nothing to say. The **device** shall perform the UNTERMINATED action described in 6.3.2.2.

The Message Exchange Control shall stay in the IDLE state when either the **dcas** or pon is TRUE. The **device** shall perform the INITIALIZE action described in 6.3.2.1.

6.3.1.2 READ State

In the READ state, the I/O Control shall read data bytes, GET, and END messages from the IEEE 488.1 bus and place them in the Input Buffer as described in 6.1.4. The Parser and Execution Control are active, and the Output Queue is empty.

The Message Exchange Control shall enter the IDLE state when either **dcas** or pon is TRUE. The **device** shall perform the INITIALIZE action described in 6.3.2.1.

The Message Exchange Control shall enter the IDLE state when each of the following conditions occur:

1) **brq** is TRUE.
2) **bay** is FALSE, and
3) **get** is FALSE.
4) **ib-empty** is TRUE.

The **device** shall perform the UNTERMINATED action described in 6.3.2.2.

NOTE — Entering the IDLE state without terminating the <RESPONSE MESSAGE> could result in a **controller** time-out when the expected data is not sent. When this happens, an *ESR? (Standard Event Status Register Query, see 10.12) can be used to determine that no data was sent because a Query Error had occurred.

The Message Exchange Control shall enter the IDLE state when the Parser sets **eom** TRUE.

The Message Exchange Control shall enter the QUERY state when the Parser sets **query** TRUE. See 6.1.6.2.3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.3.1.3 QUERY State

In the QUERY state the Parser has recognized a valid query within a <PROGRAM MESSAGE>. The I/O Control shall continue to read data bytes from the IEEE 488.1 bus and place them in the Input Buffer as described in 6.1.4.2.4. The Parser and Execution Control are active. The **device** may format its <RESPONSE MESSAGE> at this time, or it may wait until the Message Exchange Control is in the SEND or RESPONSE states, see 6.4.5.4.

The Message Exchange Control shall enter the SEND state when **brq** is TRUE and an **eom** has not been parsed. This transition indicates that the **controller** has started to read the response.

The Message Exchange Control shall enter the RESPONSE state when the Parser sets **eom** TRUE, **bav** is FALSE, **get** is FALSE, and **ib-empty** is TRUE. This transition indicates that the **device** has finished processing the Query Message and is waiting for the **controller** to read the response.

The Message Exchange Control shall enter the DEADLOCK state when all of the following Occur:

1) The I/O Control sets **bav** or **get** TRUE, indicating that the **controller** is waiting to send data to the **device**.
2) **ib-full** is TRUE.
3) **p-blocked** is TRUE.
4) **ec-blocked** is TRUE.
5) **rf-blocked** is TRUE.
6) **oq-full** is TRUE.

This transition shall cause the **device** to set the Query Error bit in the Standard Event Status Register, clear the Output Queue, and reset the Response Formatter.

The Message Exchange Control shall enter the READ state when the following conditions Occur:

1) **eom** is TRUE, and either
2) **bav** is TRUE, or
3) **get** is TRUE, or
4) **ib-empty** is FALSE.

The **device** shall perform the INTERRUPTED action described in 6.3.2.3.

The Message Exchange Control shall enter the IDLE state when either dcas or pon is TRUE. The **device** shall perform the INITIALIZE action described in 6.3.2.1.

6.3.1.4 SEND State

In the SEND state, the **controller** has started to read the <RESPONSE MESSAGE> from the **device.** The **device** continues to parse and execute the Query Message stored in the Input Buffer. The Response Formatter prepares the <RESPONSE MESSAGE> and places it in the Output Queue. The I/O Control shall send data bytes from the Output Queue to the **controller** as described in 6.1.4.2.3.

The Message Exchange Control shall enter the RESPONSE state when **eom** is TRUE, **bav** is FALSE, **get** is FALSE, and **ib-empty** is TRUE.

The Message Exchange Control shall enter the IDLE state when all of the following conditions occur:

1) **eom** is FALSE.
2) **bay** is FALSE.
3) **get** is FALSE.
4) **ib-empty** is TRUE.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The **device** shall perform the UNTERMINATED action described in 6.3.2.2.

The Message Exchange Control shall enter the READ state when any of the following conditions occur:

1) **bav** is TRUE.
2) **get** is TRUE.
3) The Parser sets **eom** TRUE and **ib-empty** is FALSE.

This condition indicates that the **controller** started sending a new message before reading all of the response. The **device** shall perform the INTERRUPTED action described in 6.3.2.3.

The **device** shall enter the IDLE state when either **dcas** or pon is TRUE. The **device** shall perform the INITIALIZE action described in 6.3.2.1.

6.3.1.5 RESPONSE State

In the RESPONSE state, the Parser has received a <PROGRAM MESSAGE TERMINATOR> and the **device** is sending the complete <RESPONSE MESSAGE> to the **controller.** The Response Formatter formats the <RESPONSE MESSAGE> and the I/O Control sends data bytes from the Output Queue to the **controller** as described in 6.1.4.2.3.

The Message Exchange Control shall enter the DONE state when the complete <RESPONSE MESSAGE>, including the <RESPONSE MESSAGE TERMINATOR>, has been sent.

The Message Exchange Control shall enter the READ state when either of the following conditions occur:

1) **bav** is TRUE.
2) **get** is TRUE.

This transition indicates that the **controller** attempted to send a new message before reading the complete <RESPONSE MESSAGE>. The **device** Shall perform the INTERRUPTED action described in 6.3.2.3.

The Message Exchange Control shall enter the IDLE state when either **dcas** or pon is TRUE. The **device** shall perform the INITIALIZE action, see 6.3.2.1.

A **device** may have nothing to say while the Message Exchange Control is in the RESPONSE state if all of the queries in the Query Message failed to generate any <RESPONSE DATA> because of syntax (Command), semantic (Execution), or device-specific errors. The **device** shall not send a <RESPONSE MESSAGE TERMINATOR> to the **controller.** The Message Exchange Control shall remain in the RESPONSE state until it either receives a data byte or GET message from the **controller** or the pon or **dcas** message becomes TRUE.

6.3.1.6 DONE State

In the DONE state, the **device** has finished sending a <RESPONSE MESSAGE> to the **controller** and is waiting for a new <PROGRAM MESSAGE> or GET from the **controller** The Parser, Execution Control, and Response Formatter are inactive and the Input Buffer and Output Queue are empty.

The Message Exchange Control shall enter the READ state and set **brq** FALSE when either

1) **bav** is TRUE, indicating that a new <PROGRAM MESSAGE> has been started, or
2) **get** is TRUE.

The Message Exchange Control shall enter the IDLE state when either **dcas** or pon is TRUE. The **device** shall perform the INITIALIZE action described in 6.3.2.1.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

6.3.1.7 DEADLOCK State

In the DEADLOCK state, the **device** has been asked to buffer more data than it has room to store. The Output Queue is full, blocking the Response Formatter, Execution Control, and Parser. The Input Buffer is full, and the **controller** is waiting to send more data bytes. to the **device.**

The Message Exchange Control breaks the deadlock by clearing the Output Queue and resetting the Response Formatter as it enters the DEADLOCK state (transition from QUERY to DEADLOCK, see 6.3.1.3). The Message Exchange Control also sets the Query Error bit in the Standard Event Status Register. While in the DEADLOCK state, the Message Exchange Control shall continue to parse and execute <PROGRAM MESSAGE UNIT> elements as usual, except that it shall discard query responses rather than place them into the Output Queue.

The Message Exchange Control shall enter the IDLE state and set **brq** FALSE when either of the following conditions occur:

1) *The Parser sets **eom** TRUE*. When the **controller** attempts to read the <RESPONSE MESSAGE>, the **device** shall not send any data bytes. Note that not sending any data bytes may cause the **controller** to time out. When this happens, an *ESR? (Standard Event Status Register Query, see 10.13.6) can be used to determine that no data was sent because a Query Error had occurred.
2) *Either **dcas** or pon is TRUE*. The **device** shall perform the INITIALIZE action described in 6.3.2.1.

6.3.2 Message Exchange Control Transition Actions

Certain transitions among the states are important enough to be named. The actions caused by the transitions are essential for reliable error detection, reporting, and recovery.

6.3.2.1 INITIALIZE Action

The INITIALIZE action is executed when the IEEE 488.2 communication channel is to be cleared.

The **device** shall perform each of the following actions:

1) Clear the Input Buffer.
2) Clear the Output Queue.
3) Reset the Parser.
4) Reset the Execution Control.
5) Reset the Response Formatter.

No error condition shall be reported.

6.3.2.2 UNTERMINATED Action

The UNTERMINATED action is executed when the **controller** attempts to read a <RESPONSE MESSAGE> from the **device** without first having sent a complete Query Message, including the <PROGRAM MESSAGE TERMINATOR>, to the **device**.

The **device** shall perform each of the following actions:

1) Set the Query Error bit in the Standard Event Status Register.
2) clear the Output Queue.
3) Optionally execute any <PROGRAM MESSAGE UNIT> elements from the incomplete message. If it executes any of the <PROGRAM MESSAGE UNIT> elements, it shall also execute all previous <PROGRAM MESSAGE UNIT> elements from the same message.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4) Discard any partially parsed <PROGRAM MESSAGE UNIT> so that the Parser will be ready to parse a new <PROGRAM MESSAGE UNIT>.
5) Set **brq** FALSE.

6.3.2.3 INTERRUPTED Action

The INTERRUPTED action is executed when the **device** is interrupted by a new <PROGRAM MESSAGE> before it finishes sending a <RESPONSE MESSAGE>.

The **device** shall perform each of the following actions:

1) Set the Query Error bit in the Standard Event Status Register.
2) Clear the Output Queue.
3) Reset the Execution Control and Response Formatter so that, when the **device** receives a new <QUERY MESSAGE UNIT>, the correct <RESPONSE MESSAGE> will be sent.
4) Set **brq** FALSE.

6.4 Protocol Rules

6.4.1 Program Message Transfer

The **device** shall exchange messages with the **controller** in accordance with IEEE 488.1 and the syntax rules of Sections 7. and 8..

The **device** shall not treat IFC or any ATN-true IEEE 488.1 command or address (except for SDC addressed to the **device**, DCL or GET) as a message terminator. The **controller** may freely intersperse ATN-true commands (except DCL, SDC, and GET) and messages addressed to other **devices** among the bytes of messages to or from the **device** without affecting the contents or interpretation of a message. Such commands may cause the **device** to suspend sending a device-te-controller message; but, when the **device** reenters TACS, the **device** shall resume sending the rest of the message. The **device** shall also be undisturbed in processing a controller-to-device message that is suspended due to the conditions mentioned. The **device** shall resume accepting the rest of a message when the **device** is again addressed to listen and ATN is set FALSE.

6.4.2 Message Source Independence

The **device** shall interpret received messages without reference to the source of the message.

Devices (for example, bus repeaters or diagnostic instruments) may record talker and listener addresses and may return such a record in response to a query, but they shall not ignore messages or interpret them differently on the basis of such information.

6.4.3 Message Exchange Sequence

The **controller** and **device** exchange complete <PROGRAM MESSAGE> and <RESPONSE MESSAGE> elements except as described under “Protocol Exceptions,” see 6.5.

Message exchanges are initiated by the **controller**. The **device** shall not generate a <RESPONSE MESSAGE> until it receives one of the following Query Messages:

1) A sequence of data bytes making up a valid <QUERY PROGRAM HEADER> that is not a macro label
2) A valid <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> that is a macro label, where the macro is defined to include a <QUERY MESSAGE UNIT>

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

3) A GET message or *TRG command, where the **device** implements the *DDT common command and the **device** trigger action defined by the *DDT command includes a <QUERY MESSAGE UNIT>
4) A GET message or *TRG command, where the **device** does not implement the *DDT common command and the **device** trigger action specified by the **device** designer is to generate a response message

See 6.1.6.2.3.

The **device** shall generate exactly one <RESPONSE MESSAGE> for each Query Message it receives. A <QUERY MESSAGE UNIT> may return one or more <RESPONSE MESSAGE UNIT> elements.

For example:

A DVM is asked to take five separate readings and return them in a single <RESPONSE MESSAGE>.

Command: **MEAS FIVE**? <PMT>

Response: 6; 2; 9; 2; 4 <RMT>

A DVM is asked to return all its setting information about its OHMS function.

Command: OHMS SET? <PMT>

Response: RANGE 100; AVGS 10;... ;AUTOZERO 1 <RMT>

Device documentation shall clearly indicate any <QUERY MESSAGE UNIT> that returns more than one <RESPONSE MESSAGE UNIT>.

The **device** shall interpret a GET message in the same way as a <PROGRAM MESSAGE> element. The **device** shall report a Command Error if it encounters a GET message between the first byte of a <PROGRAM MESSAGE> and the <PROGRAM MESSAGE TERMINATOR>.

The **controller** should not attempt to read any part of a <RESPONSE MESSAGE> from the **device** until it has sent the entire Query Message to the **device**. The **controller** should not send any part of another <PROGRAM MESSAGE> to the **device**; until it has read the entire <RESPONSE MESSAGE> from the **device**. The **device** shall detect the controller's failure to follow this protocol and shall act as described under “Protocol Exceptions,” see 6.5.

After pon or after receiving **dcas**, the **device** shall wait for a <PROGRAM MESSAGE> from the **controller**. During this time, it may assert the IEEE 488.1 SRQ (service request) signal and shall interpret and respond to IEEE 488.1 interface messages (for example, Serial Poll); but it shall not automatically generate any <RESPONSE MESSAGE> elements.

At any given time, the **device** is doing one of the following things:

1) *The **device** is idle, waiting for a message from the **controller***. The **controller** may be idle or may be communicating with some other **device(s)** on the **system bus**. (The Message Exchange Control is in the IDLE or DONE state.)
2) *The **device** is receiving a message from the **controller***. The **controller** may interrupt the transfer to perform a serial poll or to communicate with other **devices**, but should not attempt to read data bytes from the **device** until it has sent a <PROGRAM MESSAGE TERMINATOR>, see 7.5, to the **device**. (The Message Exchange Control is in the READ, QUERY, or DEADLOCK state.)
3) *The **device** is sending a <RESPONSE MESSAGE> to the controller*. The **controller** may interrupt the transfer to perform a serial poll or to communicate with other **devices**, but should not attempt to send data bytes to the **device** until it has received a <RESPONSE MESSAGE TERMINATOR>, see 8.5, from the **device**. (The Message Exchange Control is in the, SEND or RESPONSE state.)

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.4.4 Compound Queries

One or more <QUERY MESSAGE UNIT> elements can be placed in a single <PROGRAM MESSAGE>, see Section 7. Also, the **device** is allowed to generate one or more <RESPONSE MESSAGE UNIT> elements for each <QUERY MESSAGE UNIT> received. The <RESPONSE MESSAGE UNIT SEPARATOR> [semicolon (;), see 8.4.1] shall be used to separate <RESPONSE MESSAGE UNIT> elements within the complete compound <RESPONSE MESSAGE>. The <RESPONSE MESSAGE TERMINATOR>, defined in 8.5 as NL^END, shall terminate the last <RESPONSE MESSAGE UNIT>. Since the required terminator for a <RESPONSE MESSAGE UNIT> is dependent upon whether a subsequent <RESPONSE MESSAGE UNIT> will be in the same <RESPONSE MESSAGE>, the **device** is not able to terminate each <RESPONSE MESSAGE UNIT> until the Parser receives either another <QUERY MESSAGE UNIT> or a <PROGRAM MESSAGE TERMINATOR>.

Compound queries shall not span more than one complete <PROGRAM MESSAGE>. If a compound query exceeds a **device's** ability to hold the associated <RESPONSE MESSAGE> in the Output Queue, the **device's** Response Formatter, Execution Control, and Parser may become blocked waiting until room is available. If the Input Buffer becomes full while the Parser is suspended, a buffer deadlock will occur, see 6.5.7.4.

NOTE — If any of the <QUERY MESSAGE UNIT> elements of a compound query returns more than one <RESPONSE MESSAGE UNIT>, an application program cannot use the semicolon (;) delimiter to distinguish the end of the response to each <QUERY MESSAGE UNIT>.

6.4.5 Message Order Requirements

6.4.5.1 Execution Order of Program Messages

The **device** shall execute <PROGRAM MESSAGE> elements in the order received. (See 6.1.7 for definition of execute.) Execution of individual parsable elements may be done as they are received. The **device** may also wait until either a <PROGRAM MESSAGE UNIT SEPARATOR> or a <PROGRAM MESSAGE TERMINATOR> is parsed before executing preceding <PROGRAM MESSAGE UNIT> elements. The **device** shall not execute any <PROGRAM MESSAGE UNIT> before executing all prior <PROGRAM MESSAGE UNIT> elements, except as described under "Device Parameter Couplings" in 6.4.5.3. The **device** shall execute all the <PROGRAM MESSAGE UNIT> elements of a given <PROGRAM MESSAGE> before executing any <PROGRAM MESSAGE UNIT> elements of a successive <PROGRAM MESSAGE>.

6.4.5.2 Execution Order of Intermixed GET Messages and <PROGRAM MESSAGE> Elements

The **device** shall execute GET messages in the same order as <PROGRAM MESSAGE> elements. If the **device** receives a GET message while parsing or executing a prior message, execution of the GET message shall be delayed until the the prior messages have been executed.

6.4.5.3 Device Parameter Couplings

Coupled parameters are **device** functions that interact with each other in a device-specific manner. Problems can arise because the prior state of the **device** can affect the **device's** response to the programming of a coupled parameter. Ideally, the **device** shall guarantee that a complete <PROGRAM MESSAGE> containing a set of valid settings for coupled parameters will be accepted independent of the prior setting of any of the coupled parameters.

For example, a power supply can have its current and voltage set to a wide range of values with the requirement that the product of the two parameters shall be less than 100 W. Assume the voltage is set to 100 V and the current set to 1 A. The **device** should allow the application programmer to send the following message and not have an error reported.

CURRENT 100; VOLTAGE 1 NL^END

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

To accomplish sequence independence of coupled parameters, the Execution Control may buffer parsed message units and defer execution until **eom** or some device-specific command is received. The **device** may also use other techniques to achieve the same goal. The requirement that <PROGRAM MESSAGE UNIT> elements shall be executed in order of reception is relaxed under these circumstances.

If the coupled parameter <PROGRAM MESSAGE> elements are not contiguous, then the final results are unspecified. In this case, the **device** may report an Execution Error if the parameter coupling criterion is violated before the complete set of coupled parameters has been executed.

Some **devices** may have parameter couplings that are beyond the scope of the **device** parameter coupling requirements of this section.

NOTE — The **device** designer has the responsibility to ensure that **devices** incorporate effective checking to prevent execution of deferred coupled commands after an Execution Error that could result in undesireable conditions. Documentation should indicate any known conditions that cannot be checked by the **device**.

6.4.5.4 Generation of Response Message Data

Device designers may choose to generate the bytes of a <RESPONSE MESSAGE> at the time the **controller** reads a message rather than immediately following the execution of a Query Message. Sending large amounts of response data may require this technique. In such cases, the contents of the <RESPONSE MESSAGE> will represent the state of the **device** at the time the **controller** reads the response, rather than at the time the query was parsed. **Device** documentation shall indicate the query responses that will be evaluated at the time data is actually read.

When the **device** designer chooses to defer the generation of response data until **brq** is TRUE, the **device** shall set the MAV message TRUE at the point when the **device** is ready to respond to the **controller's** request for data. Thus, an application program can always rely on receiving the MAV message, via the status reporting capability, as a signal to begin a device-to-controller transfer, see 11.5.2.1.

<RESPONSE MESSAGE> elements shall be sent to the **controller** in the same order that their associated <QUERY MESSAGE UNIT> elements were sent to the **device**. Under no condition shall the **device** send the response to a <QUERY MESSAGE UNIT> before responding to a prior <QUERY MESSAGE UNIT>, except when the prior response was aborted due to one of the protocol exceptions described in 6.5.

6.5 Protocol Exceptions

6.5.1 Aborted Messages

<PROGRAM MESSAGE> and <RESPONSE MESSAGE> elements shall be aborted by dcas or **pon.**

6.5.2 Addressed to Talk With Nothing to Say

If the I/O Control sets **brq** TRUE while the Message Exchange Control is in the READ or IDLE state, the **device** has been addressed to talk with nothing to say. The **device** shall perform as indicated in 6.3.2.2.

If the I/O Control sets **brq** TRUE while the Message Exchange Control is in the RESPONSE state, but all of the queries in the Query Message failed to generate any <RESPONSE DATA> because of syntax (Command), semantic (Execution), or device-specific errors, the **device** has been addressed to talk with nothing to say. The **device** shall perform as indicated in 6.3.1.5.

NOTE — The **device** may be unable to send data when it is addressed to talk even if no error condition exists. This can occur because the **device** has deferred the preparation of the response data until an internal operation is complete or an "External Control Signal" is received. This is not an error condition and a Query Error shall not be reported. If the controller continues to wait for the data, the **device** will send it when it is available.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.5.3 No Listener on Bus

If the **device's** IEEE 488.1 Source Handshake function is in the Source Delay State (SDYS) and the Not Data Accepted (NDAC) and Not Ready For Data (NRFD) signals are beth passive false, the **device** shall not send any data bytes or take any action that would affect subsequent activity on the IEEE 488.1 bus. The **device** shall wait for a listener to assert NDAC TRUE or for the **controller** to take control. The **device** designer may report this condition to the user by means other than the **system interface**.

NOTE — This rule applies only to **devices** that are not the controller-in-charge. **Controllers** are required to report this condition to the application, see 15.3.3.

6.5.4 Command Error

A Command Error is generated under the conditions defined in 6.1.6.1.1. When the Status Reporting Control receives a Command Error, it shall set the Command Error bit in the Standard Event Status Register, see 11.5.1.1.4.

6.5.5 Execution Error

An Execution Error occurs under the conditions defined in 11.5.1.1.5. When the Execution Control detects an Execution Error, the Status Reporting Control shall set the Execution Error bit in the Standard Event Status Register. The **device** shall continue parsing the input stream. The **device** may continue executing parsed commands or the **device** may discard parsed commands. **Devices** shall resume execution of parsed commands after a <PROGRAM MESSAGE TERMINATOR>.

6.5.6 Device-Specific Error

Device-specific errors are defined in 11.5.1.1.6. These errors shall have no effect on the message exchange protocol defined in this section.

6.5.7 Query Error

Query errors are reported by setting the Query Error bit in the Standard Event Status Register as defined in 11.5.1.1.7. The **device** shall report a Query Error when the **controller** fails to follow the Message Exchange Control Protocol under any of the conditions defined in 6.3.1.7, 6.3.2.2, and 6.3.2.3.

6.5.7.1 Incomplete Command or Query Received

The device shall detect an incomplete command or query when the I/O Control sets **brq** TRUE; the Parser has received at least one byte of a <PROGRAM MESSAGE>, but not **eom**; and the Input Buffer is empty. The **device** shall respond as indicated in 6.3.2.2.

6.5.7.2 Interrupted Response

If the **device** receives a data byte or a GET message following a Query Message before the **device** has sent the <RESPONSE MESSAGE TERMINATOR>, the INTERRUPTED action shall be performed. When this happens, the **device** shall abort the rest of the response and respond as indicated in 6.3.2.3.

6.5.7.3 Query Message Units Separated by the Program Message Terminator

When an application program sends a **device** a Query Message, properly terminated with a <PROGRAM MESSAGE TERMINATOR>, and sends another <PROGRAM MESSAGE> without first reading the entire <RESPONSE MESSAGE>, the **device** shall respond as indicated in 6.3.2.3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

NOTE — If the application program sends two Query Messages without reading the response from the first, and then attempts to read the response to the second, it may receive seine data bytes from the first response followed by the complete second response. The result may or may not be a syntactically correct <RESPONSE MESSAGE>. This happens when the **system** timing allows the **controller** to send the second Query Message and begin its read operation before the Parser detects the end of the first Query Message. In this case, the Message Exchange Control enters the SEND state and starts to send the first response to the **controller**. When the Parser reaches the end of the first Query Message, the **device** is INTERRUPTED and starts to process the second query.

To avoid the above condition, application programs should not send Query Messages to a **device** without reading the associated <RESPONSE MESSAGE>. When this cannot be avoided, as in exception handling, a **device** clear should be used before a new Query Message is sent. This will assure that any fragmentary response from a prior <RESPONSE MESSAGE> will not be sent by the **device**.

6.5.7.4 Buffer Deadlock

Under some circumstances, a **device** may be unable to completely generate a <RESPONSE MESSAGE>. The **device** is deadlocked when it cannot accept another data byte of a <PROGRAM MESSAGE> from the **controller** because its Input Buffer and Output Queue are both full, and the controller cannot read the **device's** <RESPONSE MESSAGE> until it has completed sending its Query Message.

The **device** shall respond to a Buffer Deadlock as indicated in 6.3.1.7.

6.5.7.5 Query After Indefinite Response

Certain compound queries may attempt to generate an illegal <RESPONSE MESSAGE>. If a <QUERY MESSAGE UNIT> that generates an indefinite response (either an INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA or ARBITRARY ASCII RESPONSE DATA) is not the last <QUERY MESSAGE UNIT> in a <PROGRAM MESSAGE>, the **device** shall report a Query Error and not send the response from the following <QUERY MESSAGE UNIT> elements. The **device** may optionally either format or not format the response data that is discarded. Once the **device** recognizes the **eom** and sends the <RESPONSE MESSAGE TERMINATOR>, the message exchange protocol is reestablished, see 6.3.

7. Device Listening Formats

7.1 Overview

This section discusses the formatting of programming messages received by a **device** from its **system interface**. This formatting occurs on two levels. The higher level is the “functional” level at which each element has functional importance. This functional level is required for designers of the **device** command set. The lower level formatting represents the actual bus “encoding” required to transmit a functional element. This latter information is necessary for designers of **device** parsers.

7.1.1 Device Command Set Generation

Allowable IEEE 488.2 program messages are composed of sequences of program message units, each unit representing a program command or query. Each program command or query is composed of a sequence of functional syntactic elements. Legal IEEE 488.2 program commands and queries are created from functional element sequences generated by traversing the functional syntax diagram illustrated in Figs 7-1 through 7-6.

Some commands and queries of universal instrument **system** application have been defined by this standard. They are the common commands described in Section 10.. These common commands and queries are specific path selections through the functional syntax diagram. The remaining commands are device-specific and are generated by the **device**

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

designer using the functional syntax diagram and the needs of the **device**. The functional elements include separators, terminators, headers, and data types. Each element type is discussed in detail in 7.4 through 7.7.

Legal IEEE 488.2 program command structures at the functional level shall only be generated by traversing the functional syntax diagram.

7.1.2 Encoding Syntax

7.1.2.1 Allowable Syntax

A device-defined or common program command or query is sent to the **device** over the **system interface** as a sequence of data bytes. The allowable byte sequences for each functional element are defined in the encoding syntax diagram accompanying each element. Alternate, but semantically equivalent, paths are shown in each of these encoding syntax diagrams. "Data Fields" are device-definable within specified encoding limits. Allowable sequences for each functional element are generated by traversing the respective encoding syntax diagram.

For allowable functional element sequences, a **device** shall accept and correctly interpret functional elements that follow the encoding sequence. All equivalent paths, as defined for each functional element, shall be accepted by the **device's** parser.

7.1.2.2 Illegal Syntax — Command Error

A byte sequence that does not follow encoding syntax rules, as defined in 7.1.2.1, shall not be interpreted as a functional element, but shall generate a Command Error.

A functional element sequence that is not an allowable IEEE 488.2 program message, as defined in 7.1.1, shall not be interpreted as a program command or query, but shall result in the generation of a Command Error.

How Command Errors are reported is described in Section 11. Parser interaction with and recovery from Command Errors are described in Section 6.

7.1.2.3 Rationale for Encoding Syntax Flexibility

For program data, the encoding syntax diagrams are designed to allow a variety of semantically equivalent encodings of a functional element.

This flexibility is included for

1) *Human-readability*. Program commands typically appear in program code and thus need a reasonable degree of readability.
2) *Human-generation*. Program commands are typically generated by the programmer and not the **controller.** Thus, a degree of formatting flexibility is desirable to allow commonly used formatting variations.

7.2 Notation

7.2.1 Diagraming Syntactic Flow

Syntactic elements are connected by lines with directional symbols to indicate the flow. Flow through the syntax diagrams generally proceeds from left-to-right.

When an element(s) is repeatable, a reverse, right-to-left path will be shown around and above the element(s).

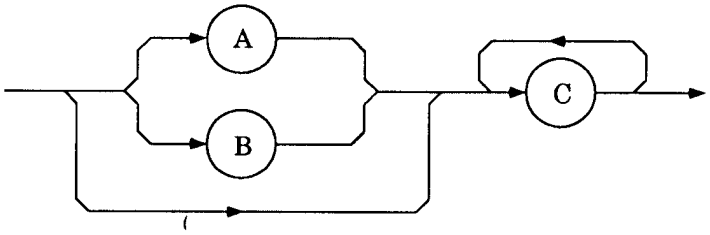
When an element(s) can be bypassed, a left-to-right path will be shown around and below the element(s).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The path branches when there is a choice of elements.

For example,



allows the following combinations of elements A, B, and C:

AC
C
BCCCCC
CCC

NOTE — The element C may be repeated indefinitely.

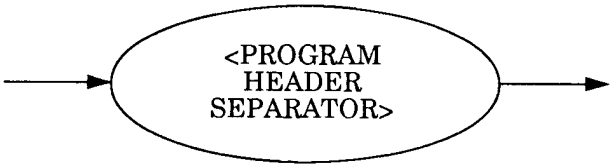
7.2.2 Syntactic Elements

7.2.2.1 Terminal Syntactic Elements

Terminal elements are the basic, indivisible syntactic constructs. They represent either a basic function in the functional syntax diagrams or a particular byte-encoding in the encoding syntax diagrams.

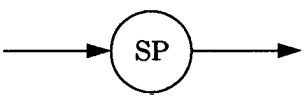
All elements in the functional syntax diagrams are represented by an all-uppercase description between brackets inside an oval.

For example,



In the encoding syntax diagrams, terminal elements are directly defined as 8 bit data bytes or by reference to the 7 bit ASCII Code Chart shown in Table 9-2. When a **device** receives 7 bit ASCII characters, the sense of the eighth bit (DI08) shall be ignored.

For example, a space is represented by the following:



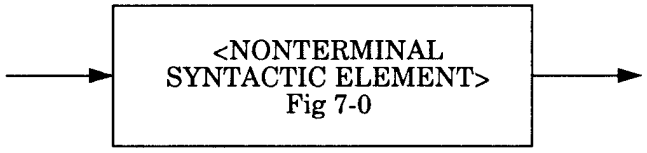
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

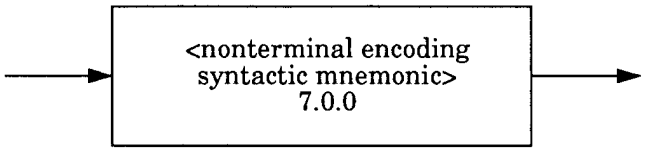
7.2.2.2 Nonterminal Syntactic Elements

Nonterminal elements are intermediate syntactic constructs presented for clarity or emphasis. Nonterminals are presented in boxes to distinguish them from terminals. A nonterminal is always expandable to a diagram of terminal elements or (for encoding syntax) is explainable in a statement relating the element to a particular code set. Below the element name is a section reference to the expansion diagram's location.

For example,



or



In the notation, functional elements are written in uppercase and encoding elements are written in lowercase between brackets, unless a particular ASCII code is indicated.

7.2.2.3 Rules

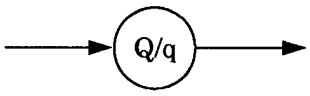
Several encoding syntax diagrams require additional text to explain restrictions not easily shown in the diagram. These restrictions are included in a “Rules” section immediately following the diagram.

7.2.3 Special Symbols

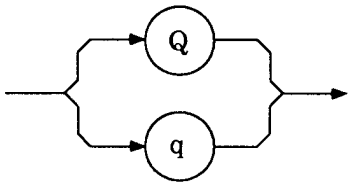
7.2.3.1 Upper/Lowercase Equivalence

To improve presentation, a special notation is used to indicate either uppercase or lowercase alternative representation. This representation is indicated by enclosing, in a circle, the uppercase ASCII alpha symbol, a slash, and the lowercase ASCII alpha symbol.

For example,



is equivalent to



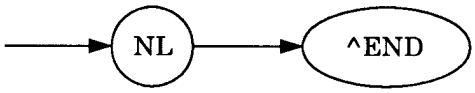
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.3.2 END Message

An oval containing "^END" indicates concurrent transmission of the END message with the preceding data byte. This symbol has no exit path as it always represents the end of a message.

For example,



represents sending NL (newline) with EOI set TRUE and ATN FALSE. In text, this is shown as NL^END.

7.2.3.3 Beginning of Message

The beginning of a message is indicated by a circle with no entry from the left.

7.2.3.4 Diagram Expansion

Leading and trailing arrows indicate that the diagram represents an expansion to a detailed view of a portion of a higher level diagram.

7.3 Terminated Program Messages — Functional Syntax

7.3.1 Function

Terminated program messages are complete "controller-to-device" messages. They are sequences of zero or more <PROGRAM MESSAGE UNIT> elements. The <PROGRAM MESSAGE UNIT> element represents a programming command or data sent to the **device** from the **controller**. The terminated message represents a "complete" transmission and, as such, has certain additional semantic meanings, see 7.5.

7.3.2 Syntax

The command's functional syntax shall match the following set of six diagrams in Figs 7-1 through 7-6.

NOTE — The use of the indefinite form of the <ARBITRARY BLOCK PROGRAM DATA> element, see 7.7.6, forces an immediate termination of the <PROGRAM MESSAGE>.

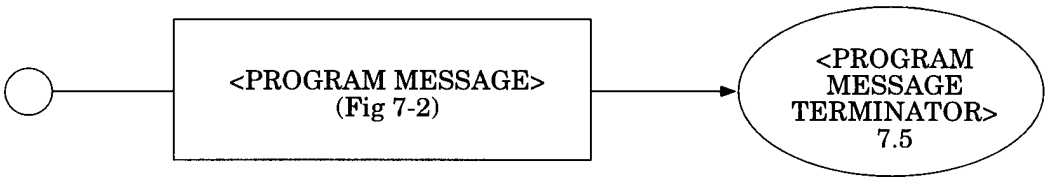


Figure 7-1—<TERMINATED PROGRAM MESSAGE> Functional Element Syntax

NOTE — Section 6.4.3 requires the **device** to interpret a GET message in the same way as a <PROGRAM MESSAGE> element. The **device** reports a Command Error if it encounters a GET message between the first byte of a <PROGRAM MESSAGE> and the <PROGRAM MESSAGE TERMINATOR>. See 6.4.3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

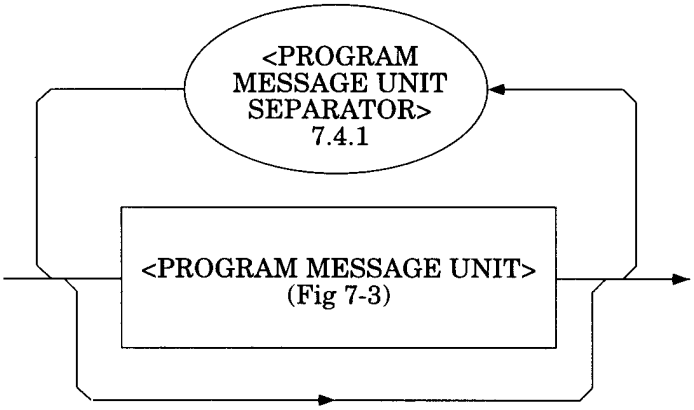


Figure 7-2—<PROGRAM MESSAGE> Functional Element Syntax

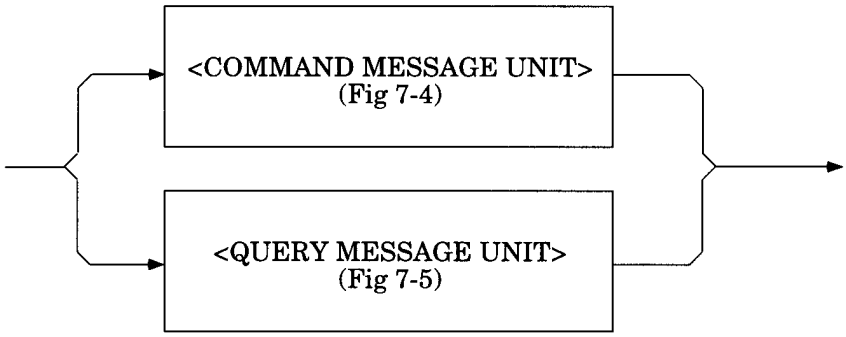


Figure 7-3—<PROGRAM MESSAGE UNIT> Functional Element Syntax

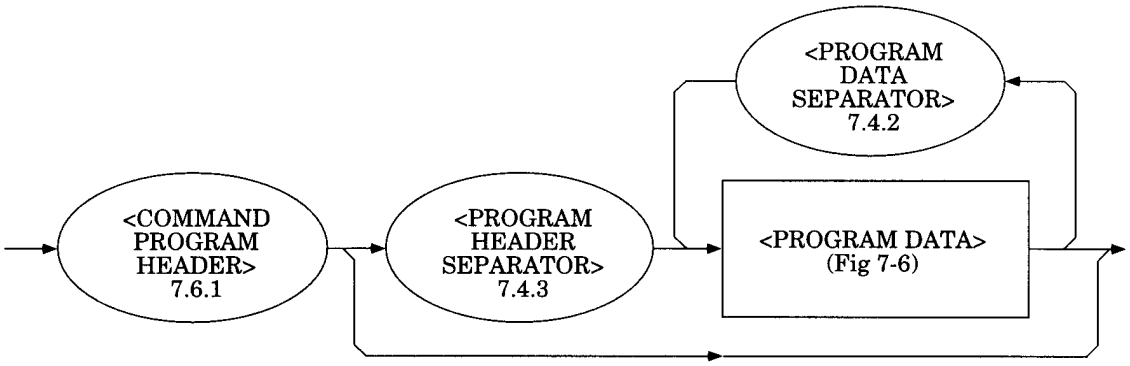


Figure 7-4—<COMMAND MESSAGE UNIT> Functional Element Syntax

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

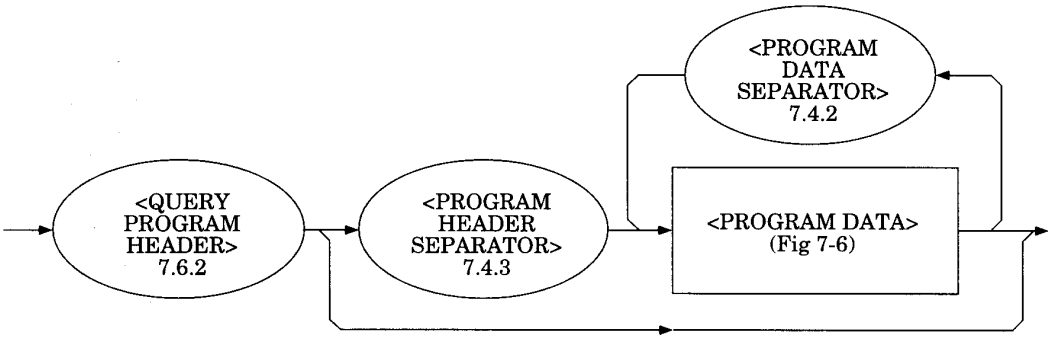


Figure 7-5—<QUERY MESSAGE UNIT> Functional Element Syntax

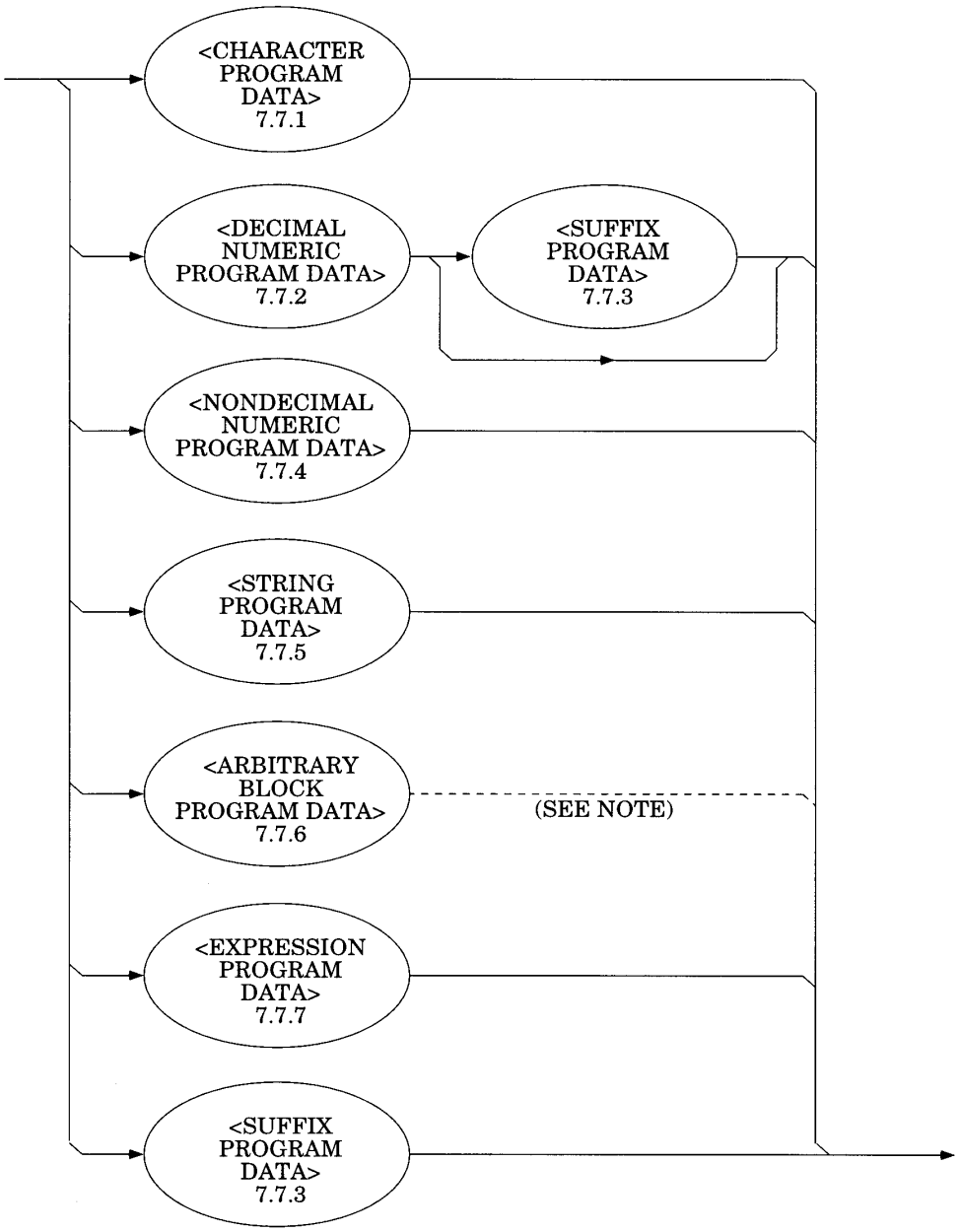


Figure 7-6—<PROGRAM DATA> Functional Element Syntax

NOTE — <ARBITRARY BLOCK PROGRAM DATA>, using the indefinite format, ends with an implicit message terminator. In this case, the message terminates with no exit. See 7.7.6.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.3.3 Functional Element Summary

Element	Function
<PROGRAM MESSAGE> Fig 7-2	Represents a sequence of zero or moro <PROGRAM MESSAGE UNIT> elements separated by <PROGRAM MESSAGE UNIT SEPARATOR> elements.
<PROGRAM MESSAGE UNIT> Fig 7-3	Represents a single command, programming data, or query received by the **device**.
<COMMAND MESSAGE UNIT> Fig 7-4	Represents a single command or programming data received by the **device**.
<QUERY MESSAGE UNIT> Fig 7-5	Represents a single query sent from the **controller** to the **device**.
<PROGRAM DATA> Fig 7-6	Represents any of the six different program data types.
<PROGRAM MESSAGE UNIT SEPARATOR> 7.4.1	Separates the <PROGRAM MESSAGE UNIT> elements from one another in a <PROGRAM MESSAGE>.
<PROGRAM DATA SEPARATOR> 7.4.2	Separates sequential <PROGRAM DATA> elements that are related to the same header.
<PROGRAM HEADER SEPARATOR> 7.4.3	Separates the header from any associated <PROGRAM DATA>.
<PROGRAM MESSAGE TERMINATOR> 7.5	Terminates a <PROGRAM MESSAGE>.
<COMMAND PROGRAM HEADER>7.6.1	Specifies function or operation. Used with any associated <PROGRAM DATA> elements(s).
<QUERY PROGRAM HEADER> 7.6.2	Similar to <COMMAND PROGRAM HEADER> except a query indicator (?) shows that a response is expected from the **device**.
<CHARACTER PROGRAM DATA> 7.7.1	A data type suitable for sending short mnemonic data, generally where a numeric data type is not suitable.
<DECIMAL NUMERIC PROGRAM DATA> 7.7.2	A data type suitable for sending decimal integers or decimal fractions with or without exponents.
<SUFFIX PROGRAM DATA> 7.7.3	A data type suitable for sending units and optional multipliers. Also an optional field following <DECIMAL NUMERIC PROGRAM DATA>.
<NONDECIMAL NUMERIC PROGRAM DATA> 7.7.4	A data type suitable for sending integer numeric representations in base 16, 8, or 2. Useful for data that is more easily interpreted when directly expressed in a nondecimal format.
<STRING PROGRAM DATA> 7.7.5	A data type suitable for sending 7 bit ASCII character strings where the content needs to be “hidden” (by delimiters).
<ARBITRARY BLOCK PROGRAM DATA> 7.7.6	A data type suitable for sending blocks of arbitrary 8 bit information.
<EXPRESSION PROGRAM DATA> 7.7.7	Data type utilizing nested parentheses with an open format. Useful for struc- tured data and parameter manipulation applications. Suitable for sending data that is evaluated as one or more nonexpression data elements before further parsing.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.4 Separator Functional Elements

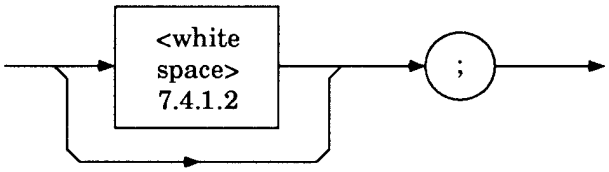
7.4.1 <PROGRAM MESSAGE UNIT SEPARATOR>

7.4.1.1 Function

The <PROGRAM MESSAGE UNIT SEPARATOR> separates sequential <PROGRAM MESSAGE UNIT> elements from one another within a <PROGRAM MESSAGE>.

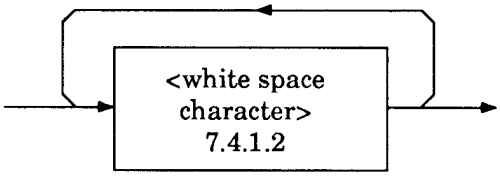
7.4.1.2 Encoding Syntax

A <PROGRAM MESSAGE UNIT SEPARATOR> is defined as



where

<white space> is defined as



where

<white-space character> is defined as a single ASCII-encoded byte in the range of 00-09, 0B-20 (0-9, 11-32 decimal). This range includes the ASCII control characters and the space but excludes the newline.

7.4.1.3 Semantic Equivalence

A **device** shall process <white space> without semantic interpretation.

A **device** shall interpret the semicolon (;) as the <PROGRAM MESSAGE UNIT SEPARATOR>. No alternative encodings are allowed.

7.4.2 <PROGRAM DATA SEPARATOR>

7.4.2.1 Function

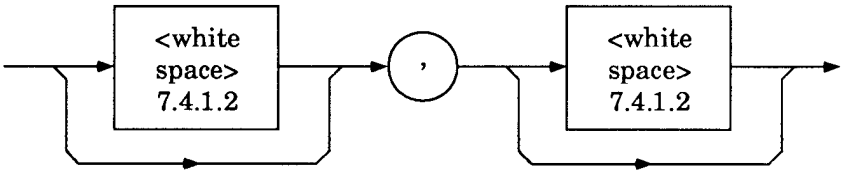
The <PROGRAM DATA SEPARATOR> separates sequential <PROGRAM DATA> elements from one another after a <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER>. It is used when a <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> has multiple parameters.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.4.2.2 Encoding Syntax

A <PROGRAM DATA SEPARATOR> is defined as



where

<white space> is defined in 7.4.1.2.

7.4.2.3 Semantic Equivalence

A **device** shall process <white space> without semantic interpretation.

A **device** shall interpret the comma (,) as the <PROGRAM DATA SEPARATOR>. No alternative encodings are allowed.

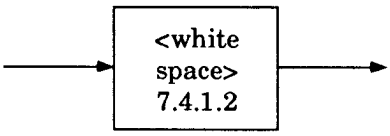
7.4.3 <PROGRAM HEADER SEPARATOR>

7.4.3.1 Function

The <PROGRAM HEADER SEPARATOR> separates the <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> from the <PROGRAM DATA> elements.

7.4.3.2 Encoding Syntax

A <PROGRAM HEADER SEPARATOR> is defined as



where

<white space> is defined in 7.4.1.2.

7.4.3.3 Semantic Equivalence

A **device** shall interpret the first <white space character> as the <PROGRAM HEADER SEPARATOR>. No alternative encodings are allowed.

Any additional <white space >shall have no syntactic nor semantic value.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.5 <PROGRAM MESSAGE TERMINATOR>

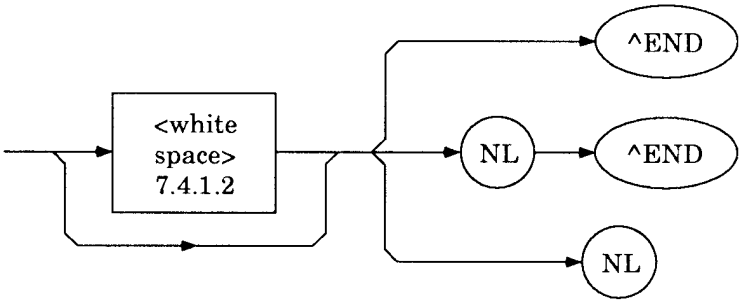
7.5.1 Function

A <PROGRAM MESSAGE TERMINATOR> terminates a sequence of one or more definite length <PROGRAM MESSAGE UNIT> elements. A <PROGRAM MESSAGE TERMINATOR> is abbreviated in this standard as <PMT>.

The terminator conveys additional semantic meaning with regard to the execution order of interactive or "coupled" commands, see 6.4.5.3, to the operation of the Output Queue, see 6.1.10, and to the syntactic use of compound headers, see 7.6.1.5 and 7.6.2.5.

7.5.2 Encoding Syntax

A <PROGRAM MESSAGE TERMINATOR> is defined as



where

<white space> is defined in 7.4.1.2.

NL is defined as a single ASCII-encoded byte 0A (10 decimal).

NOTE — An END message cannot be sent without an accompanying data byte. See IEEE Std 488.1-1987 [4]. The ^END syntactic element implies the IEEE 488.1 END message is sent with the last data byte of the preceding syntactic element.

7.5.3 Semantic Equivalence

A **device** shall process <white space> without semantic interpretation.

A **device** shall interpret any and all of the three terminators (i.e., END message on last data byte, newline with END message, or newline) as semantically equivalent. No alternative encodings are allowed.

7.6 Program Header Functional Elements

7.6.1 <COMMAND PROGPRAM HEADER>

A **device** shall not implement both <simple command program header> and <compound command program header> elements.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.6.1.1 Function

The <COMMAND PROGRAM HEADER> represents the operation to be performed in a **device**. This element may be optionally followed by associated parameters encoded as <PROGRAM DATA>. elements, see 7.7.

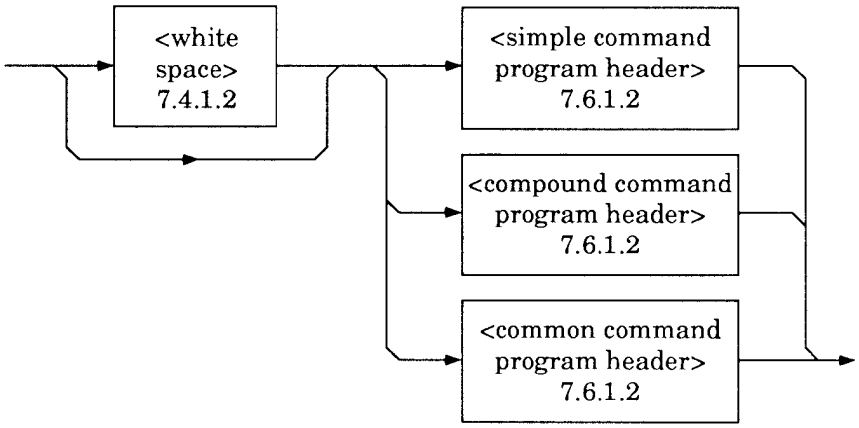
Note that a macro label (see *DMC, 10.7) may have the form of a <COMMAND PROGRAM HEADER> but may label a sequence of <PROGRAM MESSAGE UNIT> elements that contains valid <QUERY PROGRAM HEADER> elements (see 7.6.2.1), and would thus generate a response when processed. The TRG common command (see 10.37) may also be specified to generate a response.

A <compound command program header> allows for internal structure in the header. This structure is generally used by more complex **devices** to limit the number of unique headers and also to logically organize the **device** command set.

<compound command program header> elements are useful for handling hierarchically-related command structures. <compound command program header> elements can provide shorthand mnemonic notation when used within a <PROGRAM MESSAGE> containing several <PROGRAM MESSAGE UNIT> elements.

7.6.1.2 Encoding Syntax

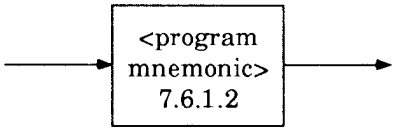
A <COMMAND PROGRAM HEADER> is defined as



where

<white space> is defined in 7.4.1.2

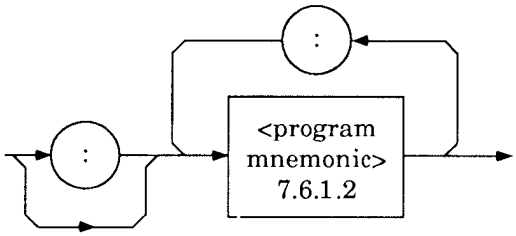
<simple command program header> is defined as



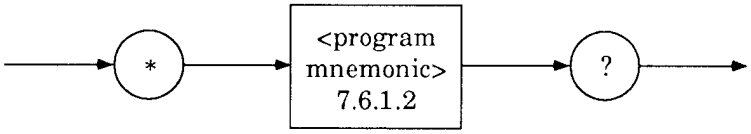
<compound command program header> is defined as

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

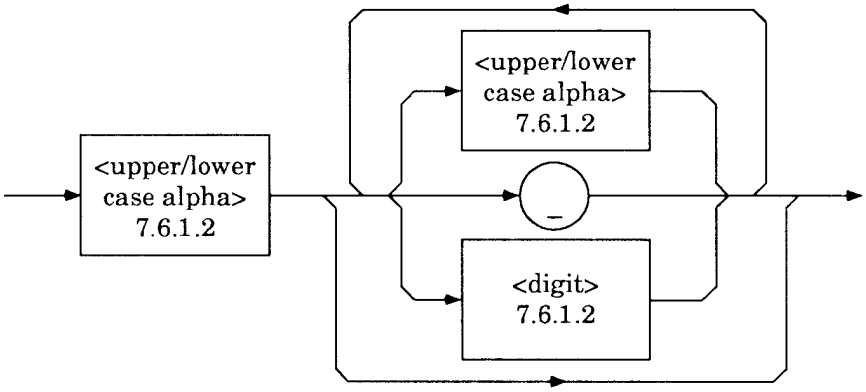
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



<common command program header> is defined as



<program mnemonic> is defined as



where

1) <upper/lowercase alpha> is defined as a single ASCII-encoded byte in the range of 41-5A, 61-7A (65-90, 97-122 decimal).
2) <digit> is defined as a single ASCII-encoded byte in the range 30-39 (48-57 decimal).
3) (_) represents an "underscore," a single ASCII-encoded byte with the value 5F (95 decimal).

7.6.1.3 Semantic Equivalence

A **device** shall

1) Interpret <upper/lowercase alpha> contained within the <program mnemonic> without attaching semantic meaning to the case of the alpha characters.
2) Interpret the embedded colon (:) as the <program mnemonic> separator in the <compound command program header> element. No alternative encodings are allowed. For semantics of leading colon, see 7.6.1.5.
3) Process <white space> contained within a <COMMAND PROGRAM HEADER> without semantic interpretation.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.6.1.4 Rules

7.6.1.4.1 Length

The <program mnemonic> shall have a maximum length of 12 characters with a preferred length of four characters.

7.6.1.4.2 <COMMAND PROGRAM HEADER> Naming Guidelines

The relation of a <COMMAND PROGRAM HEADER> mnemonic and its associated function shall be readily apparent. These mnemonics shall be independent of control location on the front panel but shall relate to front panel or display nomenclature as appropriate.

7.6.1.4.3 <common command program header> Rules

The <common command program header> syntax is reserved for use by this standard and future revisions of this standard. Defined common commands and queries using this syntax are discussed in Section 10.

Device designers shall not use the <common command program header> defined in Section 10. for purposes other than those defined in Section 10.

7.6.1.5 Header Compounding Rules

That part of a <compound command program header> that excludes the trailing <program mnemonic> element is defined as the "header-path." A <compound command program header> shall be interpreted by a **device** as if the header-path in the immediately preceding <compound command program header> or <compound query program header> was sent immediately preceding that <compound command program header>. The supplied header-path shall include both any header-path explicitly stated in the immediately preceding command and any header-path supplied to the immediately preceding command by a previous application of the rule.

This rule shall NOT be applied to:

1) A <compound command program header> that begins a <PROGRAM MESSAGE>
2) A <compound command program header> that begins with a colon (:)

NOTE — The effect of not applying this rule is that the header-path becomes null.

The presence of a <common command program header> or <common query program header> has no effect on the header-path. See Appendix A for examples. See 10.7.3 for the macro label expansion rule.

7.6.2 <QUERY PROGRAM HEADER>

A **device** shall not implement both <simple query program header> and <compound query program header> elements.

7.6.2.1 Function

The <QUERY PROGRAM HEADER> represents the operation to be performed in a **device**. A <QUERY PROGRAM HEADER> causes the **device** to generate a response. This element may be optionally followed by associated parameters encoded as <PROGRAM DATA> elements, see 7.7.

Note that a macro label (see *DMC, 10.7) may have the form of a <QUERY PROGRAM HEADER>, but may label a sequence of <PROGRAM MESSAGE UNIT> elements that contains no valid <QUERY PROGRAM HEADER> elements and, thus, would generate no response when processed.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

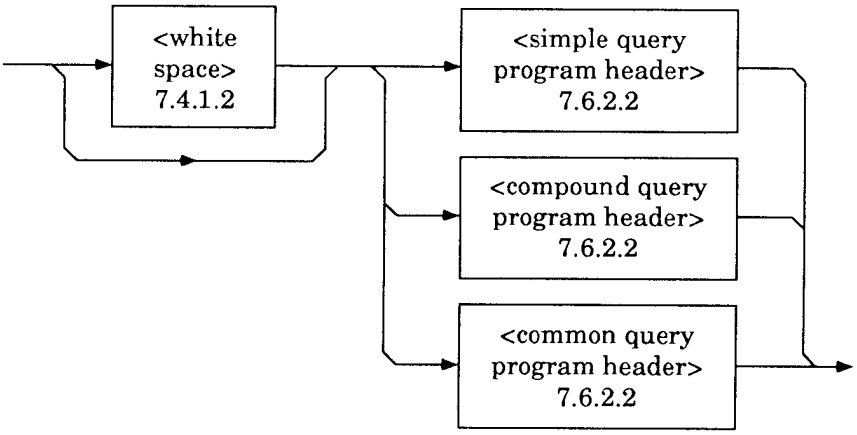
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The <compound query program header> represents the function to be performed in conjunction with possible following data elements to form a program command. A <compound query program header> allows for internal structure in the header.

This structure is generally used by more complex **devices** to limit the number of unique headers and also to logically organize the **device** command set for easier control and more-readable code. <compound query program header> elements are useful for handling command structures that are hierarchically related. <compound query program header> elements are capable of providing a shorthand mnemonic notation when used within multimessage-unit <PROGRAM MESSAGE> elements.

7.6.2.2 Encoding Syntax

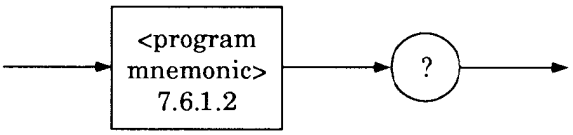
A <QUERY PROGRAM HEADER> is defined as



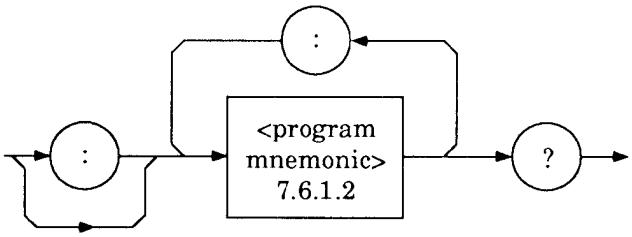
where

<white space> is defined in 7.4.1.2.

<simple query program header> is defined as



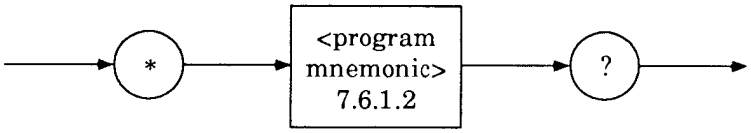
<compound query program header> is defined as



Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

<common query program header> is defined as



<program mnemonic> is defined in 7.6.1.2

7.6.2.3 Semantic Equivalence

Semantic equivalence rules for <QUERY PROGRAM HEADER> are the same as for <COMMAND PROGRAM HEADER>, see 7.6.1.3.

A **device** shall process <white space> contained within a <COMMAND PROGRAM HEADER>or <QUERY PROGRAM HEADER> without semantic interpretation.

7.6.2.4 Rules

General rules for <QUERY PROGRAM HEADER> are the same as for <COMMAND PROGRAM HEADER>, see 7.6.1.4.

7.6.2.5 Header Compounding Rules

Rules for <compound query program headers> are the same as the rules for <compound command program headers>, see 7.6.1.5. See Appendix A for examples.

7.7 <PROGRAM DATA> Functional Elements

A <PROGRAM DATA> functional element is used to convey a variety of types of parameter information related to the program header.

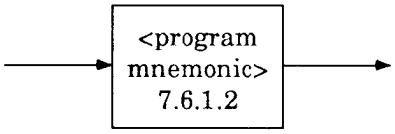
7.7.1 <CHARACTER PROGRAM DATA>

7.7.1.1 Function

The <CHARACTER PROGRAM DATA> functional element is used to convey parameter information best expressed mnemonically as a short alpha or alphanumeric string. It is useful in those cases where numeric parameters are inappropriate.

7.7.1.2 Encoding Syntax

A <CHARACTER PROGRAM DATA> element is defined as



where

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

<program mnemonic> is defined in 7.6.1.2.

7.7.1.3 Semantic Equivalence

The <program mnemonic>, as used in the <CHARACTER PROGRAM DATA> functional element, shall follow the same semantic equivalence rules as when it is used in <COMMAND PROGRAM HEADER> element, see 7.6.1.3.

7.7.1.4 Rules

The <program mnemonic>, as used in the <CHARACTER PROGRAM DATA> functional element, shall follow the same general rules as when it is used in the <COMMAND PROGRAM HEADER> element, see 7.6.1.4.

7.7.2 <DECIMAL NUMERIC PROGRAM DATA>

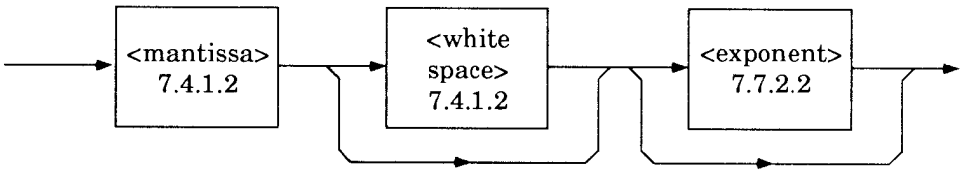
7.7.2.1 Function

The <DECIMAL NUMERIC PROGRAM DATA> is a flexible version of the three numeric representations (NR1,2, and 3) as defined in ANSI X3.42-1990 [2].

This element is also denoted by NRf (flexible numeric representation) in subsequent sections.

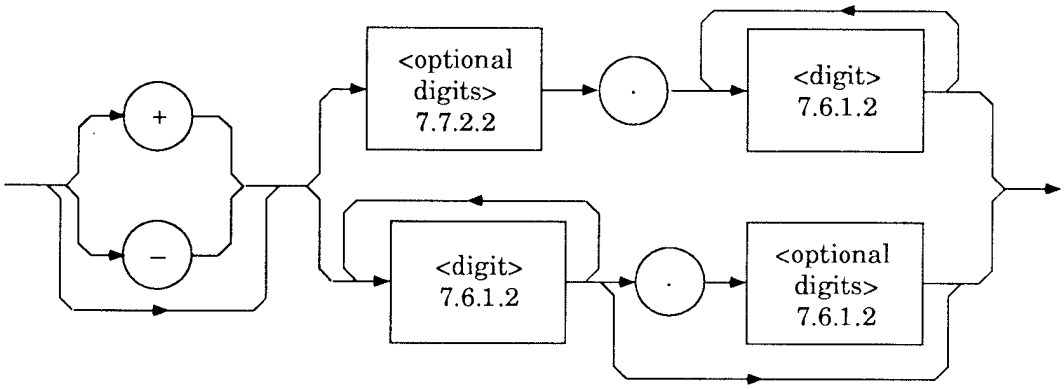
7.7.2.2 Encoding Syntax

A <DECIMAL NUMERIC PROGRAM DATA> element is defined as



where

<mantissa> is defined as

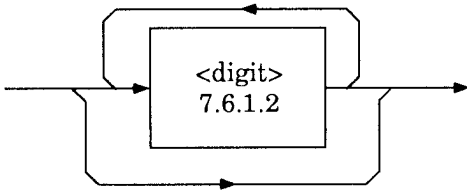


<digit> is defined in 7.6.1.2.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

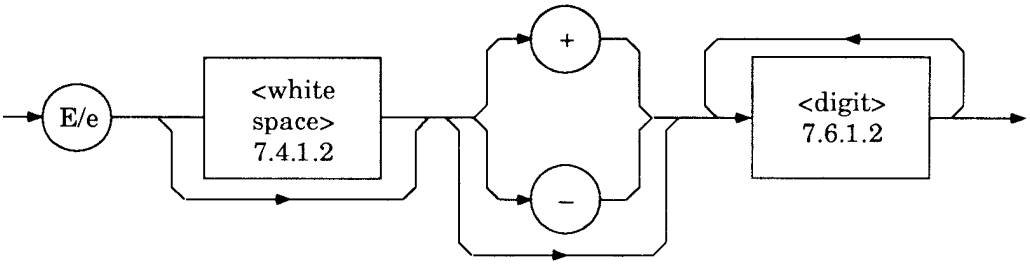
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

<optional digits> is defined as



where

<exponent> is defined as



where

<digit> is defined in 7.6.1.2.

<white space> is defined in 7.4.1.2.

7.7.2.3 Semantic Equivalence

A **device** shall process <white space> without semantic interpretation.

A **device** shall interpret all other ASCII-encoded characters as comprising a decimal numeric format generally following any of the NR1, NR2, and NR3 formats of ANSI X3. 42-1990 [2], as allowed by the diagram in 7.7.2.2. The interpreted value shall be subject to the rounding rule as stated in 7.7.2.4.2.

7.7.2.4 Rules

7.7.2.4.1 Range

The mantissa of a <DECIMAL NUMERIC PROGRAM DATA> element shall contain no more than 255 characters excluding any leading zeros.

<DECIMAL NUMERIC PROGRAM DATA> elements shall have an exponent value in the range -32000 through +32000.

If the <DECIMAL NUMERIC PROGRAM DATA> element violates either of these rules, a Command Error shall be generated.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.7.2.4.2 Numeric Element Rounding

A **device** may receive a <DECIMAL NUMERIC PROGRAM DATA> element that has greater precision than the **device** can handle internally. In this case, the **device** shall round the number, not truncate the number, before interpretation. When rounding, the **device** shall ignore the sign of the number and round up on values greater than or equal to exact half values. Values less than exact half values shall be rounded down.

7.7.2.4.3 Error Reporting

Any error conditions reported shall be based on the result of the interpreted value after rounding.

7.7.2.4.4 Numeric Out-of-Range Guidelines

If a <DECIMAL NUMERIC PROGRAM DATA> element's value is outside the range allowed for the associated header, an Execution Error shall be reported.

7.7.2.4.5 Infinity and Not-a-Number Representations

It is recommended that positive infinity be represented as 9.9E+37 and negative infinity as -9.9E+37. It is also recommended that not-a-number be represented by 9.91E+37.

These numeric values were chosen to fit into a single 32 bit IEEE 754 floating point number, see 9.3.2.1. **Devices** employing 64 bit double format numbers with a greater exponent range should take care to ensure that the above numbers are outside of the range of real numeric response data.

7.7.3 <SUFFIX PROGRAM DATA>

7.7.3.1 Function

A <SUFFIX PROGRAM DATA> element is used to express units and (optional) multipliers. Where <SUFFIX PROGRAM DATA> is used following a <DECIMAL NUMBERIC PROGRAM DATA> (NRf), the suffix expresses how the preceding NRf is interpreted by the **device.**

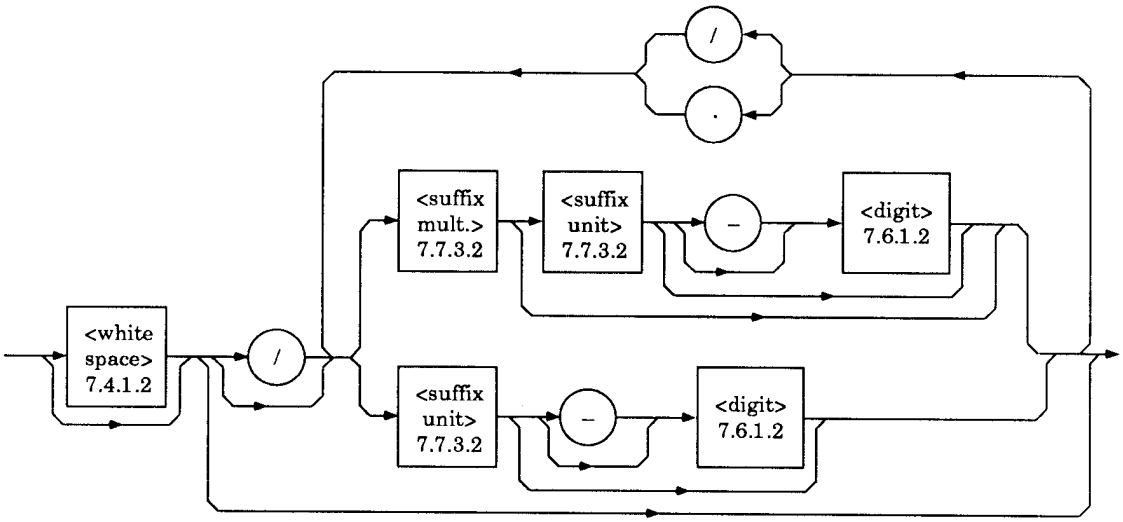
As shown in the functional syntax in Fig 7-6, the presence of a <SUFFIX PROGRAM DATA> after an NRf is always optional. No particular <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> in a **device** shall require the use of a <SUFFIX PROGRAM DATA> element.

7.7.3.2 Encoding Syntax

A <SUFFIX PROGRAM DATA> element is defined as

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



where

1) <digit> is defined in 7.6.1.2.
2) <suffix mult.>, suffix multiplier, is defined as a sequence of one, two, or three <upper/ lowercase alpha> characters.
3) <suffix unit> is defined as a sequence of one to four <upper/lowercase alpha> characters. <upper/lowercase alpha> is defined in 7.6.1.2.

7.7.3.3 Semantic Equivalence

A **device** shall process <white space> without semantic interpretation.

A **device** shall interpret all alpha characters with upper/lowercase semantic equivalence.

All other ASCII-encoded bytes in this functional element may or may not have semantic equivalence at the discretion of the **device** designer.

The bypass within the <SUFFIX PROGRAM DATA> element shall be interpreted by the **device** as if a default <suffix multiplier> and <suffix unit> had been received. This default suffix is specified by the **device** designer. The default suffix unit is to be mentioned explicitly in the **device** documentation.

7.7.3.4 Rules

If a **device** uses suffixes, it should make use of the <suffix multiplier> and <suffix unit> elements defined in this section. Guidelines presented here for the use of <SUFFIX PROGRAM DATA> elements have been defined in ISO 2955 : 1983 [19] and have been expanded to include non-SI derived units in IEEE Std 260-1978 [3].

Any <SUFFIX PROGRAM DATA> element not specifically listed in this standard shall contain mnemonic value to assist the user in relating the associated conversion to the suffix.

Numeric rounding may be done before or after the conversion algorithm is applied.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A <SUFFIX PROGRAM DATA> element shall contain no more than twelve characters, excluding the leading <white space>. All units that are defined or identified by ISO 1000 :1992 [18] are included in Table 7-1.

7.7.3.4.1 <suffix unit> Selection

Table 7-1 presents some of the most common <suffix unit> elements used in **devices** along with several preferred and allowed secondary <suffix unit> elements for a variety of applications.

<suffix unit> elements should use the "primary unit" mnemonics from Table 7-1. Alternate <suffix unit> elements, however, are included in Table 7-1 as "secondary units" for use by the casual user who tends to think in measurement units. If unit is not specified, the referenced standards or ISO 31 :1992 [6]—[17] should be consulted.

The use of "E<digit>" or "E-<digit>" as a <SUFFIX PROGRAM DATA> element or as the leading characters of a <SUFFIX PROGRAM DATA> element is specifically disallowed because of possible confusion with the exponent of the preceding NRf.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7-1—<suffix unit> Elements

Quantity	Preferred Suffix (Primary Unit)	Allowed Suffix (Secondary Unit)	Referenced Unit
Absorbed dose	GY		Gray
Activity of a radionuclide	BQ		Becquerel
Amount of substance	MOL		Mole
Angle			
— plane		DEG	Degree
— plane		GON	Grade
— plane		MNT	Minute
— plane	RAD		Radian
— plane		SEC	Second
— plane		REV	Revolution
— solid	SR		Steradian
Dose equivalent	SV		Sievert
Electric			
— Capacitance	F		Farad
— Charge	C		Coulomb
— Conductance	SIE		Siemens
— Current	A		Ampere
— Potential (diff.)	V		Volt
— Resistance	OHM		Ohm
		MOHM*	Megaohm
Energy			
—		EV	Electron-volt
—	J		Joule
Force	N		Newton
Frequency	HZ		Hertz
		MHZ	Megahertz
Illuminance	LX		Lux
Inductance	H		Henry
Length			
—		ASU	Astronomic unit
—		IN	Inch
—		FT	Foot
—	M		Meter
—		PRS	Parsec
—		MI	Mile
—		NAMI	Nautical mile
Luminous Flux	LM		Lumen
Luminous Intensity	CD		Candela
Magnetic			

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Quantity	Preferred Suffix (Primary Unit)	Allowed Suffix (Secondary Unit)	Referenced Unit
— Flux	WB		Weber
— Induction	T		Tesla
Mass			
— Atomic	U		Atomic mass unit
—		G	Gram†
—		TNE	Tonne
Power			
—	W		Watt
—	DBM	DBMW	dBs ref to 1 mW
Pressure			
—		ATM	Atmosphere
—		INHG	Inch of mercury
—		MMHG	mm of mercury
—	PAL		Pascal
—		TORR	Tort
— (Fluid)	BAR		Bar
Ratio‡			
—	DB		Decibel
—	PCT		Percent
—		PPM	Parts per million
Temperature			
— Celsius	CEL		Degree Celsius
— Fahrenheit		FAR	Degree Fahrenheit
— Thermodynamic	K		Kelvin
Time			
—	S		Second
—		D	Day
—		HR	Hour
—		MIN	Minute
—		ANN	Year
Viscosity			
— Kinematic	ST		Strokes
— Dynamic	P		Poise
Volume	L		Liter

*Because of accepted industry practice, in combination with the suffix unit OHM and HZ, M may be used instead of MA as a suffix multiplier for mega. In the case of OHM and HZ, the use of M for milli is disallowed to prevent a potential confusion between mego and milli. This practice does not follow ISO 2955 :1983 [19]. Neither suffix, MOHM or MHZ, may be used with a suffix multiplier.

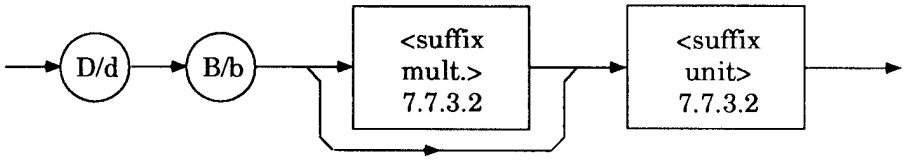
†The kilogram (KG) is the SI unit for mass.

‡Ratios are inherently unitless. Several terms are used to make the values associated with ratios more tractable to the user.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.7.3.4.2 DB referencing

To reference a relative unit (dB) to an absolute level, the absolute level ia appended to the DB according to the following syntax.



Thus, decibels referenced to 1 mW becomes DBMW. For historical reasons, DBM is allowed as an alias for DBMW.

For example,

DBUW means DB referenced to 1 μW.

DBUV means DB referenced to 1 μV.

NOTE — A provision in the appendix of IEEE Std 260-1978 {3} states that when a DB unit is referenced, the reference should be placed in parentheses. A search of common practice shows that the form proposed here is the common usage and does not introduce ambiguity.

7.7.3.4.3 <suffix mult.> Selection Rules

The <suffix mult.> is used to modify the value of the <suffix unit>. The <suffix mult.> allows the user to enter values in common measurement units. Thus, in some applications, 1 kHZ is preferable to 1E3 HZ. Table 7-2 lists the allowed <suffix mult.>'s.

NOTE: Only engineering unit multipliers are allowed.

7.7.3.4.4 Use of Units Made Up of Combinations of SI Rules

When a need arises for complex units (that is, newton-meter or meter per second) the following rules should apply. When the complex unit is the quotient of two units, then the character "/", should be used to separate the units. Thus, M/S is the allowed representation for meter/second.

When the complex unit is the product of two units, the character "." should be used to separate the units. Thus, N.M is the allowed representation for newton-meter.

Suffix exponents are represented by a <digit>. They are used for units that appear more than once in a complex unit. When a suffix exponent is used, it should appear as a <digit> and should be appended to the unit without a space. An example is the SI unit of acceleration, meters per second squared, written as M/S2.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7-2—Allowed <suffix mult.> Mnemonics

Definition	Mnemonic
1E18	EX
1E15	PE
1E12	T
1E9	G
1E6	MA (see Note)
1E3	K
1E-3	M (see Note)
1E-6	U
1E-9	N
1E-12	P
1E-15	F
1E-18	A

NOTES:

1 — The suffix units, MHZ and MOHM, are special cases that should not be confused with <suffix mult.> HZ or <suffix muir.> OHM. These special case <suffix units> are included in Table 7-6 because of accepted industry practice.

2 — In most cases, a range of <suffix mult.> and <suffix unit> mnemonic combinations can be freely picked from Tables 7-6 and 7-1. There are, however, inherent ambiguities between the two tables, i.e., Femto and Farads have the same mnemonic, as do Atto and Amperes, Kilo and Degree Kelvin, Tera and Tesla, Giga and Gram, Mega and Meter, and nano and Newton. A method of resolving these ambiguities is to design the **device** to accept only selected <suffix mult.> <suffix unit> combinations based on the requirements of the particular command.

7.7.3.4.5 Use as an Entire <PROGRAM DATA ELEMENT>

Where <SUFFIX PROGRAM DATA> elements are used without an accompanying <DECIMAL NUMERIC PROGRAM DATA> element, the allowed contents shall be the same as those allowed for <SUFFIX PROGRAM DATA> elements used in conjunction with a <DECIMAL NUMERIC PROGRAM DATA> element. The <SUFFD[PROGRAM DATA> element sets the default for <DECIMAL NUMERIC PROGRAM DATA> and <DECIMAL NUMERIC RESPONSE DATA>. This element shall not be considered an extension to <CHARACTER PROGRAM DATA>.

7.7.4 <NONDECIMAL NUMERIC PROGRAM DATA>

7.7.4.1 Function

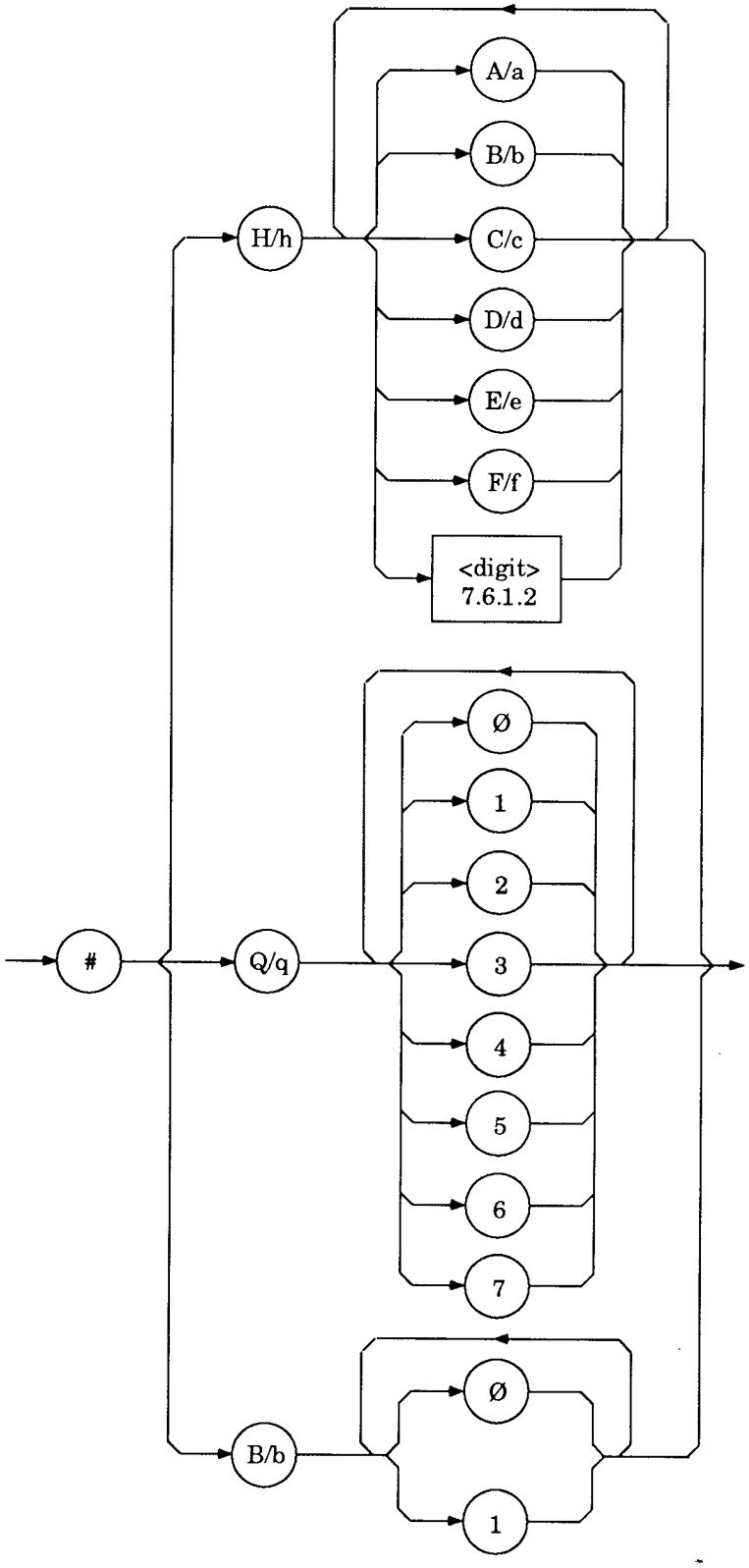
The <NONDECIMAL NUMERIC PROGRAM DATA> element allows passing numeric information in bases other than ten. These formats facilitate direct interpretation by **system** users and application programs for those applications where nondecimal numbers have direct significance. The use of this format may simplify conversion routines in application software.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.7.4.2 Encoding Syntax

A <NONDECIMAL NUMERIC PROGRAM DATA> element is defined as



Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

where

<digit> is defined in 7.6.1.2.

7.7.4.3 Semantic Equivalence

A **device** shall interpret any and all of the three specified nondecimal numeric formats as semantically equivalent numeric values.

A **device** shall interpret <upper/lowercase alpha> without attaching semantic meaning to the case of the alpha characters.

7.7.4.4 Rules

7.7.4.4.1 Hexadecimal Encoding

The characters following a #H or #h preamble shall be interpreted as an unsigned implicit radix point hexadecimal number.

The radix for hexadecimal numbers is 16. The decimal values and related ASCII code representations are as follows:

Decimal Value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ASCII Code:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Alternate ASCII Code:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Nonhex characters shall generate a Command Error.

7.7.4.4.2 Octal Encoding

The characters following a #Q or #q preamble shall be interpreted as an unsigned implicit radix point octal number.

The radix for octal numbers is eight. The decimal values and related ASCII code representations are as follows:

Decimal Value:	0	1	2	3	4	5	6	7
ASCII Code:	0	1	2	3	4	5	6	7

Nonoctal characters shall generate a Command Error.

7.7.4.4.3 Binary Encoding

The characters following a #B or #b preamble shall be interpreted as an unsigned implicit radix point binary number.

The radix for binary numbers is two. The decimal values and related ASCII code representations are as follows:

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Decimal Value:	0	1
ASCII Code:	0	1

Nonbinary characters shall generate a Command Error.

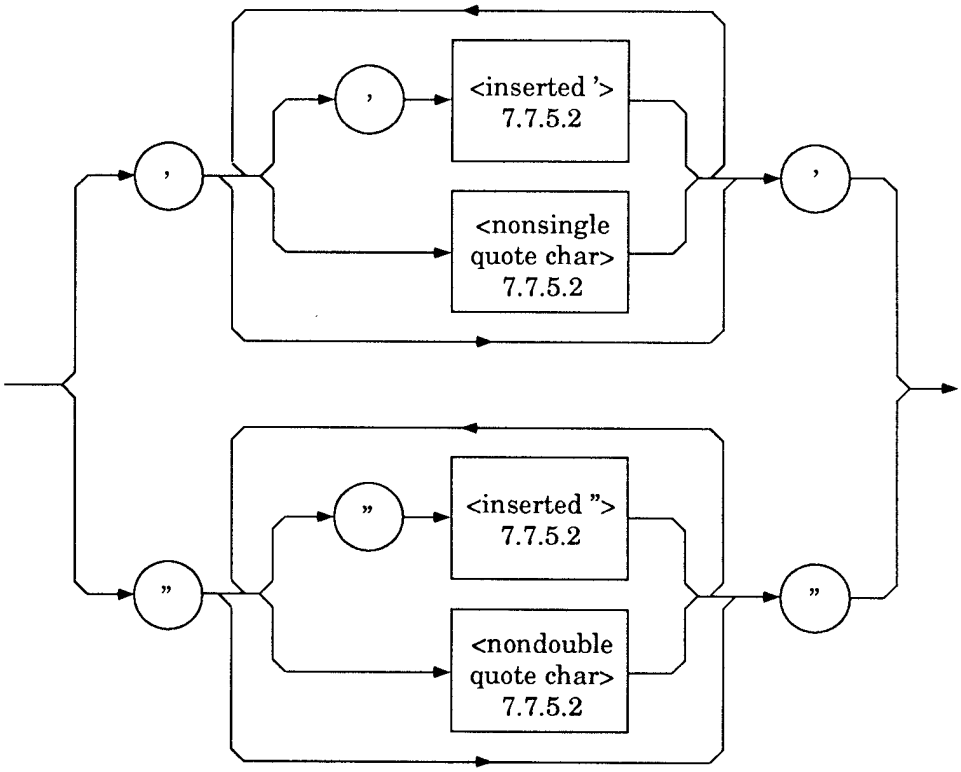
7.7.5 <STRING PROGRAM DATA>

7.7.5.1 Function

The <STRING PROGRAM DATA> element allows any character in the ASCII 7 bit code (including nonprintable characters) to be transmitted as a message. This data field is particularly useful where text is to be displayed (for example, on a printer or CRT type **device**). The <STRING PROGRAM DATA> permits the use of format effectors, such as carriage return, newline, or space to correctly format text.

7.7.5.2 Encoding Syntax

A <STRING PROGRAM DATA> element is defined as



where

1) <inserted '> is defined as a single ASCII character with the value 27 (39 decimal).
2) <nonsingle quote char> is defined as a single ASCII character of any value except 27 (29 decimal).
3) <inserted "> is defined as a single ASCII character with the value 22 (34 decimal).
4) <nondouble quote char> is defined as a single ASCII character of any value except 22 (34 decimal).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

7.7.5.3 Semantic Equivalence

A **device** shall interpret string data encapsulated in either of the two specified delimited-string formats (that is, the use of double quotes or single quotes) as semantically equivalent.

7.7.5.4 Rules

7.7.5.4.1 Single Quote Delimiting

A single quote (') followed directly by an <inserted '> is used to represent a single' within a <STRING PROGRAM DATA> delimited by single quotes.

7.7.5.4.2 Double Quote Delimiting

A double quote (“) followed directly by an <inserted “> is used to represent a single ” within a <STRING PROGRAM DATA> delimited by double quotes.

7.7.6 <ARBITRARY BLOCK PROGRAM DATA>

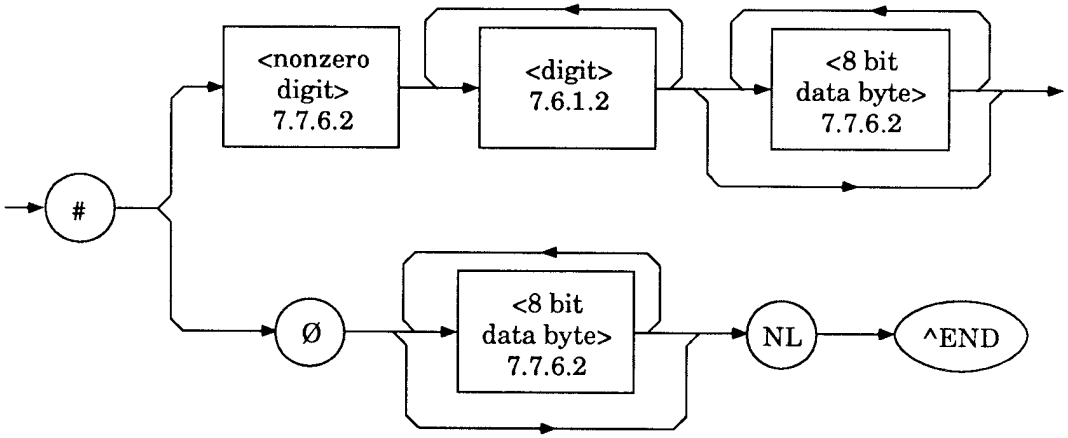
7.7.6.1 Function

The <ARBITRARY BLOCK PROGRAM DATA> element allows any 8 bit bytes (including extended ASCII codes) to be transmitted in a message. This element is particularly useful for sending large quantities of data.

This element represents a general purpose solution to the transmission of 8 bit binary data. Specific binary coding is described in 9.2 and 9.3.

7.7.6.2 Encoding Syntax

An <ARBITRARY BLOCK PROGRAM DATA> element is defined as



where

1) <digit> is defined in 7.6.1.2.
2) <nonzero digit> is defined as a single ASCII encoded byte in the range of 31-39 (49-57 decimal).
3) <8 bit data byte> is defined as an 8 bit byte in the range of 00-FF (0-255 decimal).

NOTE — The use of the indefinite format (#0) requires NL^END and forces immediate termination of the <PROGRAM MESSAGE>. Receipt of an END message with a DAB that is not an NL generates a Comand Error. NL^END constitutes the <PROGRAM MESSAGE TERMINATOR> and is not a part of the <PROGRAM MESSAGE>.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.7.6.3 Semantic Equivalence

A **device** shall interpret <8 bit data bytes> in either of the two specified formats (that is, with #<nonzero digit> or #0 with the NL^END message termination) as semantically equivalent.

7.7.6.4 Rules

The value of the <nonzero digit> element shall equal the number of <digit> elements that follow. The value of the <digit> elements taken together as a decimal integer shall equal the number of <8 bit data byte> elements that follow. If the IEEE 488.1 END message is received before the specified number of bytes has been received, a Command Error shall be reported.

7.7.6.5 Notes and Examples

Designers should consider the implications of buffer size and availability, particularly in the intended receiving **devices**. The designer should not assume that all **devices** will be capable of accepting lengthy messages. See 4.9 for **device** specification requirements.

The indefinite format is useful when the length of the transmission is not known or when transmission speed or other considerations prevent segmentation into definite length blocks.

For example, four data bytes (DABs) may be sent using:

1) #14<DAB><DAB><DAB><DAB>
2) #3004<DAB><DAB><DAB><DAB>
3) #0<DAB><DAB><DAB><DAB> NL^END

If the sequence:

#14<DAB><DAB><DAB><DAB^END>

is received, the last DAB is part of the block. The associated END message also terminates the <PROGRAM MESSAGE>. If the last DAB happened to be an NL, it is still part of the block and not part of the <PROGRAM MESSAGE TERMINATOR>.

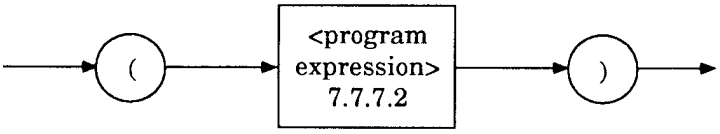
7.7.7 <EXPRESSION PROGRAM DATA>

7.7.7.1 Function

The <EXPRESSION PROGRAM DATA> element evaluates to a scalar, vector, matrix, or string value. It allows parameters to be manipulated by the **device**.

7.7.7.2 Encoding Syntax

An <EXPRESSION PROGRAM DATA> element is defined as



where

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

<program expression> is defined as:

1) A sequence of ASCII-encoded data bytes in the range 20 to 7E (32 to 126 decimal) except the double quote, number sign, single quote, left parenthesis, right parenthesis, and semicolon (characters 22, 23, 27, 28, 29, and 3B hexidecimal or 34, 35, 39, 40, 41, and 59 decimal respectively).
2) A device-defined set of the <PROGRAM DATA> elements described in 7.7, except the indefinite form of the <ARBITRARY BLOCK PROGRAM DATA>. This allows the representation of nested expressions.

7.7.7.3 Semantic Equivalence

A **device** shall interpret <upper/lowercase alpha> contained in the <program expression> element without attaching semantic meaning to the case of the alpha characters.

ASCII-encoded bytes contained in the <program expression> element may have other rules or conditions for semantic equivalence at the discretion of the **device** designer.

7.7.7.4 Rules

The <program expression> semantics are completely device-dependent, but the **device** shall be able to transform the <program expression> into the equivalent of one or more <PROGRAM DATA> elements that are not <program expression> elements.

Devices may allow properly nested subexpressions within an <program expression> up to a device-defined maximum nesting depth. A subexpression is a syntactically correct <program expression> that is completely contained within an <program expression>.

The **device** documentation shall clearly indicate which <PROGRAM DATA> elements may appear with an <program expression> as well as the maximum subexpression nesting depth. The **device** documentation shall clearly indicate any additional syntax restrictions that the **device** may place on the <program expression>.

The characters listed in 7.7.7.2 (double quote, number sign, single quote, left and right parentheses, and semicolon) may not appear within the body of the <program expression>, but may appear within a <STRING PROGRAM DATA> or <ARBITRARY BLOCK PROGRAM DATA> element in an <program expression> or subexpression. In addition, left and right p arentheses may appear in a <program expression> as part of an included subexpression that includes one or more <EXPRESSION PROGRAM DATA> elements.

8. Device Talking Elements

8.1 Overview

This section discusses the formatting of <RESPONSE MESSAGE> elements sent from a **device** via its **system interface**. As in Section 7., the formatting occurs on two levels: the functional level and the encoding level. The overview material for Section 7. applies here, except that the codes are **device,** and not human, generated and thus embody a minimum amount of flexibility in the encoding syntax. For human readability and consistency, the encoding syntax is, in most cases, obtained from a subset of the controller-to-device syntax.

Particular needs may require a **device** to talk directly, i.e., not through a **controller**, to a **device** or class of **devices** that do not understand the IEEE 488.2 <RESPONSE MESSAGE> syntax. For example, a **device** may be able to calibrate itself by commanding a particular model source to supply a stimulus. A more common example might be a **device** that is able to use the bus to command a printer or plotter to print or graph internally stored data. All deviations from IEEE 488.2 to achieve device-to-device message transfer, as described earlier, shah be minimized and shall be explicitely documented.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.2 Notation

Syntax diagram notation follows the same format as is described in 7.2.

8.3 Terminated Response Messages — Functional Syntax

8.3.1 Function

A terminated <RESPONSE MESSAGE> is a device-to-controller message. It typically contains measurement results, settings, or status information.

A <RESPONSE MESSAGE> is interpreted by a **controller** running an application program and, as such, needs to convey its information precisely for consistent operation with a wide range of **controllers**.

A <RESPONSE MESSAGE>, therefore, has a more restrictive format than a <PROGRAM MESSAGE>, see Section 7.

8.3.2 Syntax

The functional syntax of the **device's** response shall be generated by following the four-diagram set in Figs 8-1 through 8-4. The use of either the <ARBITRARY ASCII RESPONSE DATA> element (see 8.7.11) or the <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element (see 8.7.10) forces an immediate termination of the <RESPONSE MESSAGE>.

The <RESPONSE MESSAGE UNIT> has two basic syntaxes. The first is a "precise" version of the <PROGRAM MESSAGE UNIT> syntax, and is generally used for returning setting information. The second is "headerless" and is typically used to return measurement results with a minimum of bus overload.

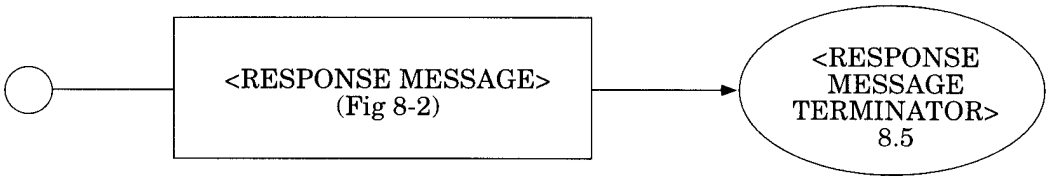


Figure 8-1—<TERMINATED RESPONSE MESSAGE> Functional Element Syntax

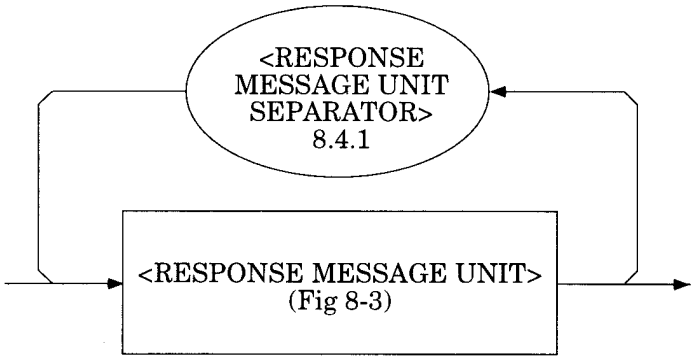


Figure 8-2—<RESPONSE MESSAGE> Functional Element Syntax

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

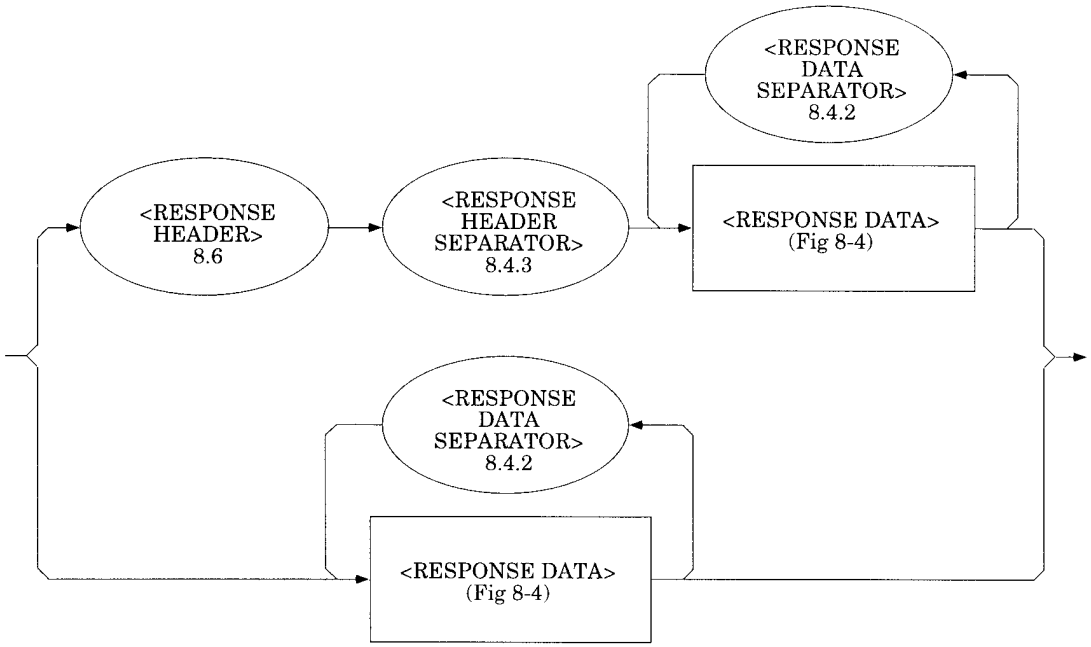


Figure 8-3—<RESPONSE MESSAGE UNIT> Functional Element Syntax

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

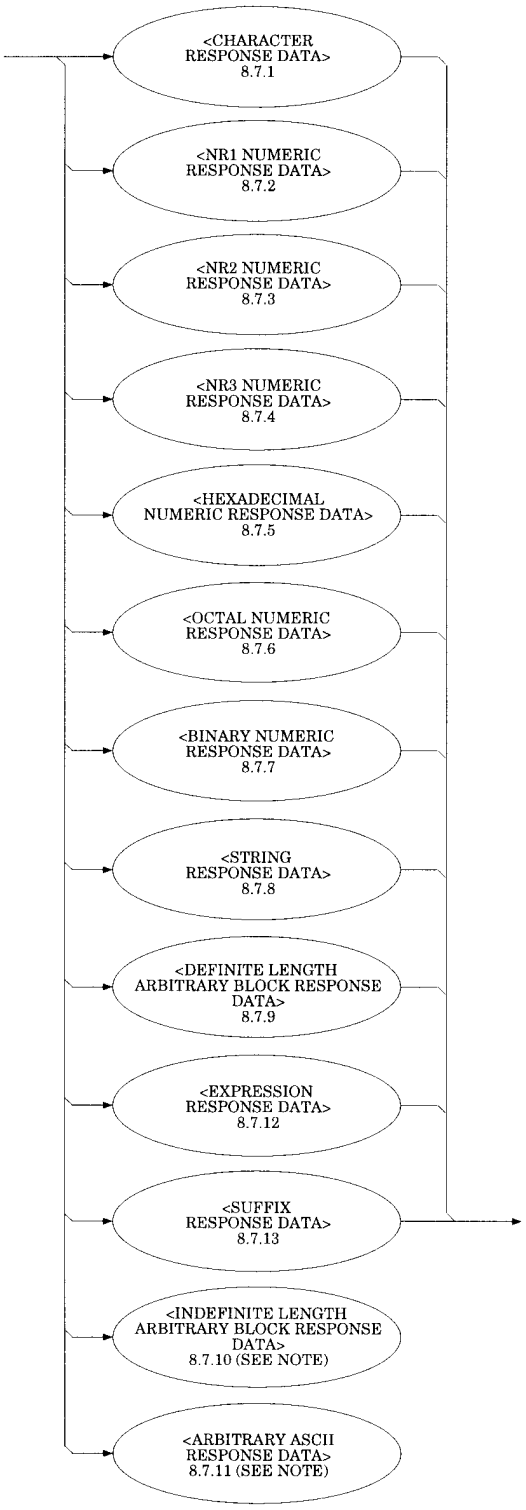


Figure 8-4—<RESPONSE DATA> Functional Element Syntax

NOTE — The <ARBITRARY ASCII RESPONSE DATA> and <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> elements end with an implied terminator. In this case, the message terminates with no exit.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.3 Functional Element Summary

Element	Function
<RESPONSE MESSAGE> Fig 8-2	Represents a sequence of one or more <RESPONSE MESSAGE UNIT> elements separated by <RESPONSE MESSAGE UNIT SEPARATOR> elements.
<RESPONSE MESSAGE UNIT> Fig 8-3	Represents a single message unit sent from the **device**.
<RESPONSE DATA> Fig 8-4	Represents one of the thirteen defined <RESPONSE DATA> elements.
<RESPONSE MESSAGE UNIT SEPARATOR> 8.4.1	Separates the <RESPONSE MESSAGE UNIT> elements from one another in a <RESPONSE MESSAGE>.
<RESPONSE DATA SEPARATOR> 8.4.2	Separates sequential <RESPONSE DATA> elements that are related to the same header or to each other.
<RESPONSE HEADER SEPARATOR> 8.4.3	Separates the header from the <RESPONSE DATA>.
<RESPONSE MESSAGE TERMINATOR>8.5	Terminates a <RESPONSE MESSAGE>.
<RESPONSE HEADER> 8.6	Specifies the function of the associated<PROGRAM DATA> elements(s). Alpha and numeric characters mnemonically indicate the function. Internal structure is provided for hierarchical header structuring.
<CHARACTER RESPONSE DATA>8.7.1	A data type suitable for sending short mnemonic character strings, generally when a numeric parameter is not suitable.
<NR1 NUMERIC RESPONSE DATA> 8.7.2	Suitable for sending implicit radix decimal values.
<NR2 NUMERIC RESPONSE DATA> 8.7.3	Suitable for sending explicit radix decimal values.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Element	Function
<NR3 NUMERIC RESPONSE DATA> 8.7.4	Suitable for sending explicit radix decimal values with an exponent.
<HEXADECIMAL NUMERIC RESPONSE DATA> 8.7.5	Suitable for sending implicit radix hexadecimal values.
<OCTAL NUMERIC RESPONSE DATA>8.7.6	Suitable for sending implicit radix octal values.
<BINARY NUMERIC RESPONSE DATA>8.7.7	Suitable for sending implicit radix binary values.
<STRING RESPONSE DATA>8.7.8	A data type suitable for sending ASCII character strings in which the contents needs to be "hidden" by delimiters. This element is generally used to send data for direct display on a **device**.
<DEFINITE" LENGTH ARBITRARY BLOCK RESPONSE DATA>8.7.9	A data type suitable for sending blocks of 8 bit binary information when the length is known beforehand.
<INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>8.7.10	A data type suitable for sending blocks of 8 bit binary information when the length is not known beforehand or when computing the length beforehand is undesireable.
<ARBITRARY ASCII RESPONSE DATA>8.7.11	Suitable for sending arbitrary ASCII bytes when alternate data types are unworkable.
<EXPRESSION RESPONSE DATA>8.7.12	Data type utilizing nesting parentheses with an open format. Useful for structured data and parameter manipulation applications. Suitable for sending data that is evaluated as one or more nonexpression data elements before further parsing.
<SUFFIX RESPONSE DATA> 8.7.13	A data type suitable for sending a unit and multiplier if present.

8.4 Separator Functional Elements

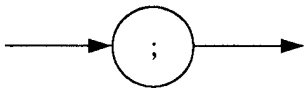
8.4.1 <RESPONSE MESSAGE UNIT SEPARATOR>

8.4.1.1 Function

The <RESPONSE MESSAGE UNIT SEPARATOR> separates sequential <RESPONSE MESSAGE UNIT> elements from one another when multiple <RESPONSE MESSAGE UNIT> elements are sent in a <RESPONSE MESSAGE>.

8.4.1.2 Encoding Syntax

A <RESPONSE MESSAGE UNIT SEPARATOR> is defined as



Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

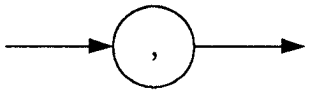
8.4.2 <RESPONSE DATA SEPARATOR>

8.4.2.1 Function

The <RESPONSE DATA SEPARATOR> separates sequential <RESPONSE DATA> elements from one another when multiple data elements are sent.

8.4.2.2 Encoding Syntax

A <RESPONSE DATA SEPARATOR> is defined as



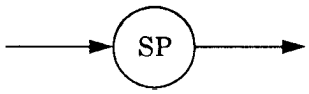
8.4.3 <RESPONSE HEADER SEPARATOR>

8.4.3.1 Function

The <RESPONSE HEADER SEPARATOR> unambiguously separates the <RESPONSE HEADER> from the <RESPONSE DATA>.

8.4.3.2 Encoding Syntax

A <RESPONSE HEADER SEPARATOR> is defined as



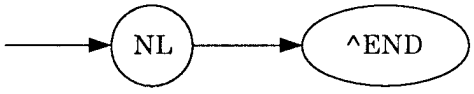
8.5 <RESPONSE MESSAGE TERMINATOR>

8.5.1 Function

The <RESPONSE MESSAGE TERMINATOR> element's function is to terminate a sequence of one or more <RESPONSE MESSAGE UNIT> elements.

8.5.2 Encoding Syntax

A <RESPONSE MESSAGE TERMINATOR> is defined as



Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

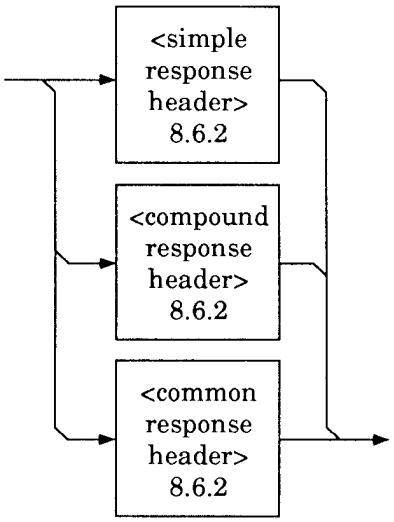
8.6 <RESPONSE HEADER>

8.6.1 Function

The <RESPONSE HEADER> is available for use by the **device** designer for device-specific responses. It may be used, for example, to create responses in directly resendable <PROGRAM MESSAGE UNIT> format or to identify response data to the controller.

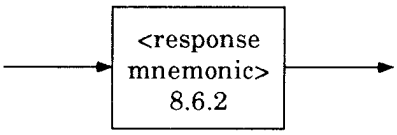
8.6.2 Encoding Syntax

The <RESPONSE HEADER> is defined as

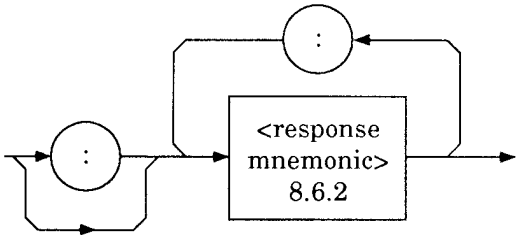


where

<simple response header> is defined as



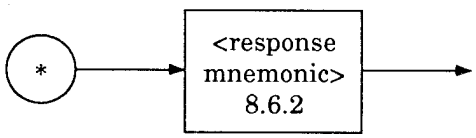
<compound response header> is defined as



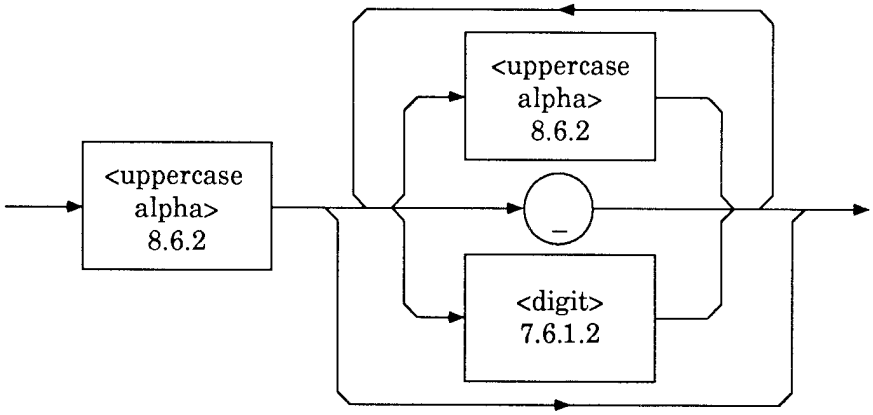
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

<compound response header> is defined as



<response mnemonic> is defined as



where:

<uppercase alpha> is defined as a single ASCII-encoded byte in the range of 41-5A (65-90 decimal).

<digit> is defined in 7.6.1.2.

(_) represents an "underscore," a single ASCII-encoded byte with the value 5F (95 decisreal).

8.6.3 Rules

The <response mnemonic> shall have a maximum length of 12 characters.

The relation of a <response mnemonic> and its related function shall be readily apparent.

The semantics of <compound response mnemonic> elements used in a <RESPONSE HEADER> is beyond the scope of this standard.

<common response header> elements may only be used to represent <COMMAND PROGRAM HEADER> elements defined in Section10..

8.7 <RESPONSE DATA> Functional Elements

A <RESPONSE DATA> functional element conveys a variety of response information. The element type is determined by the eliciting query or query equivalent (for example, macro expansion).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

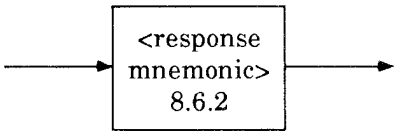
8.7.1 <CHARACTER RESPONSE DATA>

8.7.1.1 Function

The <CHARACTER RESPONSE DATA> functional element is used to convey information best expressed mnemonically as a short alpha or alphanumeric string. It is useful when numeric parameters are inappropriate.

8.7.1.2 Encoding Syntax

A <CHARACTER RESPONSE DATA> is defined as



where

<response mnemonic> is defined in 8.6.2.

8.7.1.3 Rules

The <response mnemonic> shall comply with the rules as stated in 7.6.1.4.1-7.6.1.4.3.

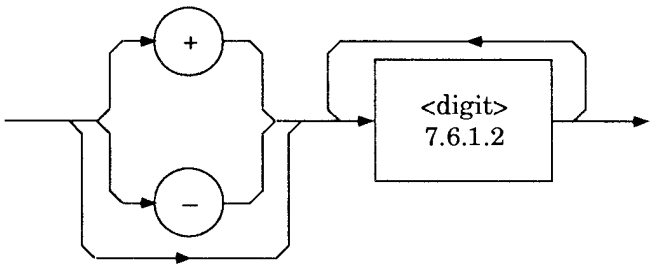
8.7.2 <NR1 NUMERIC RESPONSE DATA>

8.7.2.1 Function

<NR1 NUMERIC RESPONSE DATA> elements consist of a set of implicit point representations of numeric values. That is, a radix point is implicitly considered to be placed (fixed but not transmitted) at the right end of the string of digits.

8.7.2.2 Encoding Syntax

An <NR1 NUMERIC RESPONSE DATA> is defined as



Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.2.3 Rules

8.7.2.3.1 Out-of-Range

Numeric representation of out-of-range information shall be sent with the same numeric data type as for in-range information with a value outside of the normal range of measurements. Also, the DDE bit shall be set in the Standard Event Status Register. See 11.5.1.1.6.

For example, a voltmeter with a maximum measurement capability of 100 V could indicate that it was measuring above 100 V by returning a response with a predefined "overrange" value above 100. This value shall be returned as <NR1 NUMERIC RESPONSE DATA> if the normal in-range responses are returned as <NR1 NUMERIC RESPONSE DATA> elements.

8.7.2.3.2 Allowable Range

The allowable range for NRls is the same as the range for double precision floating point numbers (+/-179.76...E+306). See ANSI X3.42-1990 [2] and Section 9. for details of floating point formats.

Default Suffix

8.7.2.3.3 Default Suffix

If the <N**Default Suffix**RESPONSE DATA> describes a **device** setting that can be programmed using a <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> that allows the use of an optional <SUFFIX PROGRAM DATA> element, then the implied units and multiplier of the <NR1 NUMERIC RESPONSE DATA> should be the same as those defined by the defau< suffix associated with the <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER>, see 7.7.3.3. Documentation of the defau< suffix semantics or response messages is recommended.

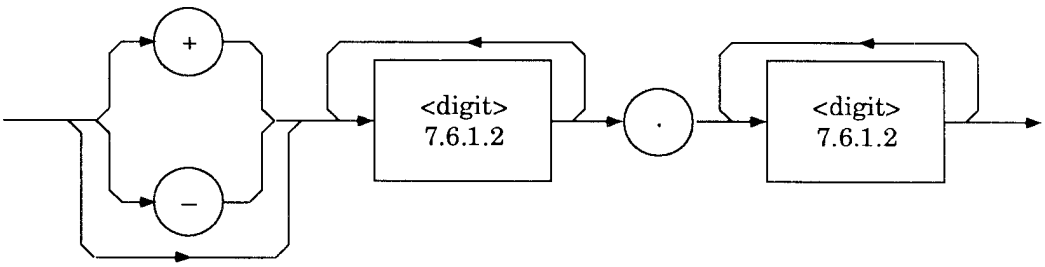
8.7.3 <NR2 NUMERIC RESPONSE DATA>

8.7.3.1 Function

<NR2 NUMERIC RESPONSE DATA> elements are the representations of explicit point numeric values.

8.7.3.2 Encoding Syntax

An <NR2 NUMERIC RESPONSE DATA> is defined as



where

<digit> is defined in 7.6.1.2.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.3.3 Rules

8.7.3.3.1 Out-of-Range

Out-of-range encoding rules are identical to those of 8.7.2.3.

8.7.3.3.2 Allowable Range

The allowable range for NR2s is the same as the range for double precision floating point numbers (+/-179.76...E+306). See ANSI X3.42-1990 [2] and Section 9. for details of floating point formats.

8.7.3.3.3 Default Suffix

The recommendation for NR2 default suffix usage is the same as described in 8.7.2.3.3.

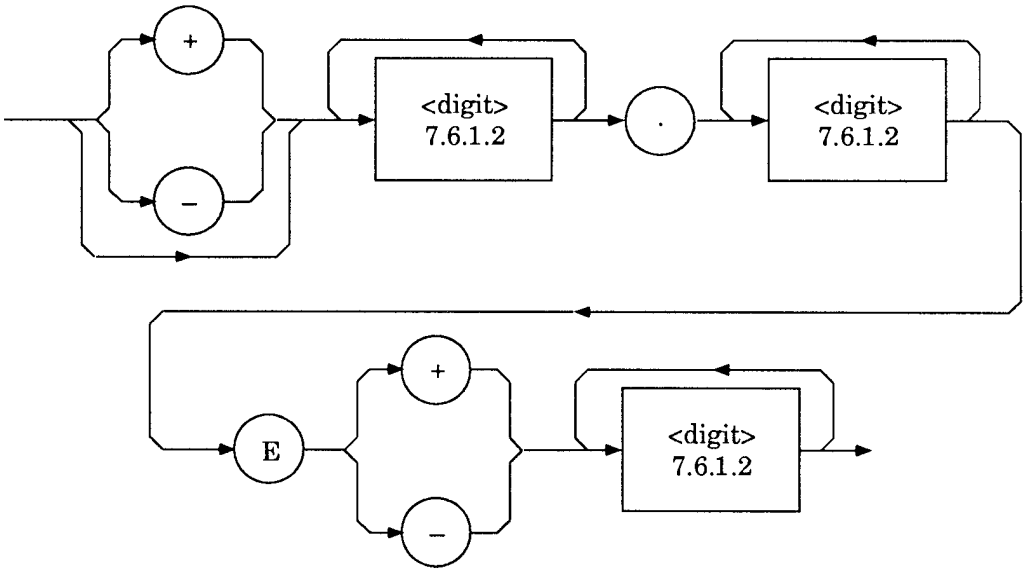
8.7.4 <NR3 NUMERIC RESPONSE DATA>

8.7.4.1 Function

<NR3 NUMERIC RESPONSE DATA> elements are representations of scaled explicit radix point numeric values together with an exponent notation.

8.7.4.2 Encoding Syntax

An <NR3 NUMERIC RESPONSE DATA> is defined as



where

<digit> is defined in 7.6.1.2.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.4.3 Rules

8.7.4.3.1 Out-of-Range

Out-of-range encoding rules are identical to those of 8.7.2.3.

8.7.4.3.2 Allowable Range

The allowable range for NR3s is the same as the range for double precision floating point numbers (+/-179.76...E+306). See ANSI X3.42-1990 [2] and Section 9. for details of floating point formats.

8.7.4.3.3 Default Suffix

The recommendation for NR3 default suffix usage is the same as described in 8.7.2.3.3.

8.7.4.4 Infinity and Not-a-Number Representations

It is recommended that positive infinity be represented as 9.9E+37 and negative infinity as -9.9E+37. It is also recommended that not-a-number be represented by 9.91E+37.

These numeric values were chosen to fit into a single 32 bit IEEE 754 floating point number, see 9.3.2.1. **Devices** employing 64 bit double format numbers with a greater exponent range should take care to ensure that the above numbers are outside of the range of real numeric response data.

8.7.5 <HEXADECIMAL NUMERIC RESPONSE DATA>

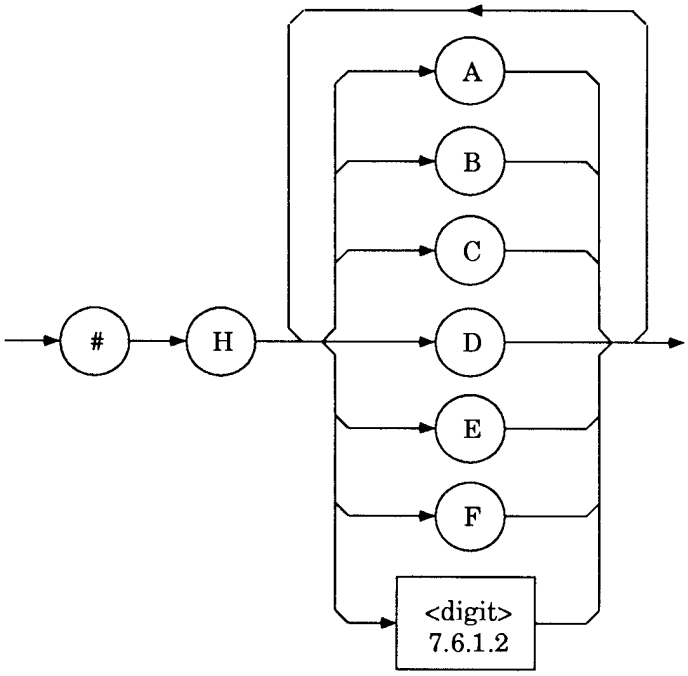
8.7.5.1 Function

The <HEXADECIMAL NUMERIC RESPONSE DATA> element is used to represent implicit radix, base 16 numeric information.

8.7.5.2 Encoding Syntax

A <HEXADECIMAL NUMERIC RESPONSE DATA> is defined as

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



where

<digit> is defined in 7.6.1.2.

8.7.5.3 Rules

The encoding rules are identical to those in 7.7.4.4.1, except that the alternate (lowercase) ASCII code option is disallowed.

8.7.6 <OCTAL NUMERIC RESPONSE DATA>

8.7.6.1 Function

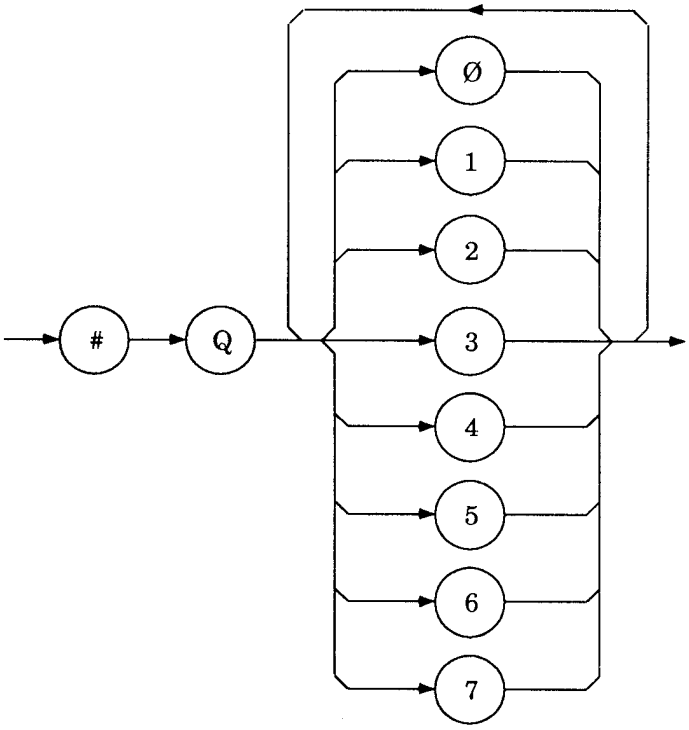
The <OCTAL NUMERIC RESPONSE DATA> element is used to represent base eight, implicit radix numeric information.

8.7.6.2 Encoding Syntax

An <OCTAL NUMERIC RESPONSE DATA> is defined as

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



8.7.6.3 Rules

The encoding rules are identical to those in 7.7.4.4.2, except that the alternate (lowercase) ASCII code option is disallowed.

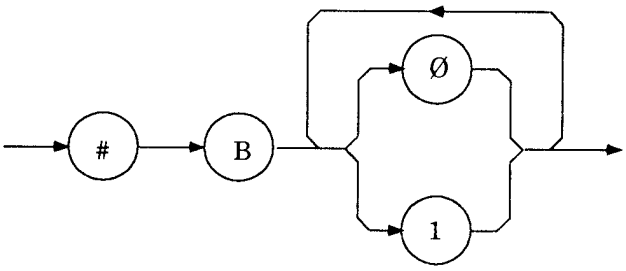
8.7.7 <BINARY NUMERIC RESPONSE DATA>

8.7.7.1 Function

The <BINARY NUMERIC RESPONSE DATA> element is used to represent base two, implicit radix numeric information.

8.7.7.2 Encoding Syntax

A<BINARY NUMERIC RESPONSE DATA> is defined as



Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.7.3 Rules

The encoding rules are identical to those in 7.7.4.4.3, except that the alternate (lowercase) ASCII code option is disallowed.

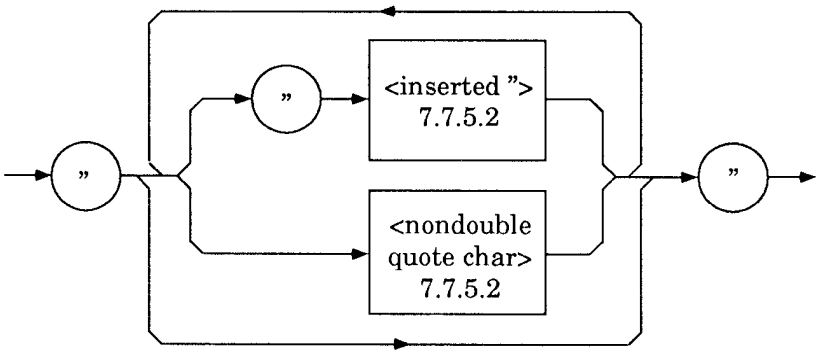
8.7.8 <STRING RESPONSE DATA>

8.7.8.1 Function

The <STRING RESPONSE DATA> elements allow any character in the ASCII 7 bit code (including nonprintable characters) to be transmitted as part of a message. This data field is particularly useful when text is to be displayed (for example, on a printer or CRT-type **device**). The <STRING RESPONSE DATA> element permits the use of format effectors such as carriage return, newline, or space to correctly format text.

8.7.8.2 Encoding Syntax

A <STRING RESPONSE DATA> is defined as



where

<inserted “> is defined in 7.7.5.2.

<nondouble quote char> is defined in 7.7.5.2.

8.7.8.3 Rules

The encoding rules are the same as for <STRING PROGRAM DATA>, see 7.7.5.4.

Bit 8 shall be set false when transmitting 7 bit ASCII data.

8.7.8.4 Notes and Examples

Alternate methods of encapsulating ASCII text, limited to response from a **device**, are the <ARBITRARY ASCII RESPONSE DATA> element, see 8.7.11, the <DEFINITE LENGTH: ARBITRARY BLOCK RESPONSE DATA> element, see 8.7.9, and the <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element, see 8.7.10. The latter element should only be used if alternate formats are unworkable.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.9 <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>

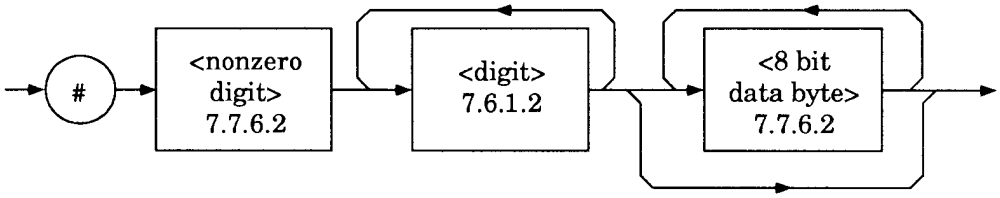
8.7.9.1 Function

The <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element allows any type of device-dependent data to be transmitted over the system interface as a series of 8 bit data bytes. This element is particularly useful for sending large quantities of data, 8 bit extended ASCII codes, or other data that are not directly displayable.

Specific recommended binary encodings for use with this element are described in Section 9.

8.7.9.2 Encoding Syntax

A <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> is defined as



where

<nonzero digit> is defined in 7.7.6.2.

<digit> is defined in 7.6.1.2.

<8 bit data byte> is defined in 7.7.6.2.

8.7.9.3 Rules

The rules are identical to those of 7.7.6.4.

8.7.9.4 Notes and Examples

Device designers should consider the implications of buffer size and availability ill the intended receiving **controllers.** Designers should not assume that all **controllers** will automatically be capable of accepting lengthy messages. See Section 14. for **controller** specification requirements.

8.7.10 <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>

8.7.10.1 Function

The <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element is useful when transmitting 8 bit data bytes when the length of the transmission is not known or where transmission speed or other conditions prevent segmenting the output into known length blocks.

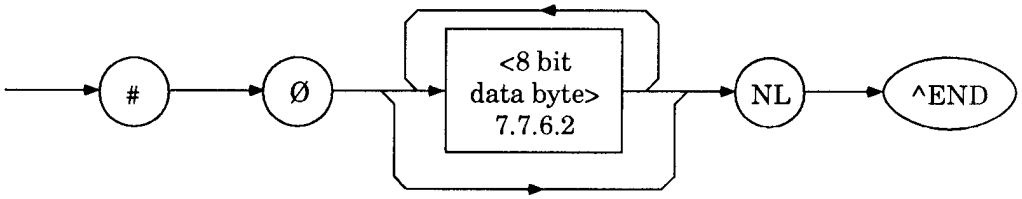
The functional syntax diagrams permit the use of this element only as the final element in a <TERMINATED RESPONSE MESSAGE>, see Fig 8-4.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.10.2 Encoding Syntax

An <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> is defined as



where

<8 bit data byte> is defined in 7.7.6.2.

NOTE — The IEEE 488.1 END message serves the dual function of terminating this element as well as terminating the <RESPONSE MESSAGE>. It is only sent once with the last byte of the indefinite black data. The NL is present for consistency with the <RESPONSE MESSAGE TERMINATOR>.

8.7.10.3 Rules

This format should only be used where other <RESPONSE DATA> element formats are unworkable.

8.7.11 <ARBITRARY ASCII RESPONSE DATA>

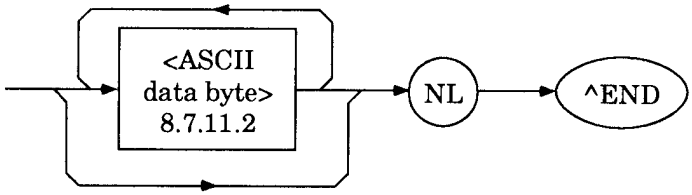
8.7.11.1 Function

The <ARBITRARY ASCII RESPONSE DATA> element allows **devices** to respond with undelimited 7 bit ASCII text. In some cases, this element facilitates **controller** handling of **device** information, such as displayed text.

The functional syntax diagrams permit the use of this element only at the end of a <TERMINATEDRESPONSE MESSAGE>, see Fig 8-4.

8.7.11.2 Encoding Syntax

An <ARBITRARY ASCII RESPONSE DATA> is defined as



where

<ASCII data byte> represents any ASCII-encoded data byte except NL (0A, 10decimal).

Notes:

1—The END message provides an unambiguous termination to an element that contains arbitrary ASCII characters

Copyright The Institute of Electrical and Electronics Engineers, Inc. ce from IEEE. © 1992 IEEE. All rights reserved.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

2— The IEEE 488.1 END message serves the dual function of terminating this element as well as terminating the <RESPONSE MESSAGE>. It is only sent once with the last byte of the indefinite block data. The NL is present for consistency with the <RESPONSE MESSAGE TERMINATOR>.

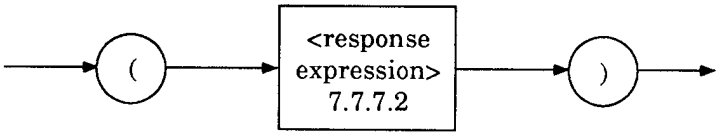
8.7.12 <EXPRESSION RESPONSE DATA>

8.7.12.1 Function

The <EXPRESSION RESPONSE DATA> element has the same scope and function as the <EXPRESSION PROGRAM DATA> element described in 7.7.7. This element allows <EXPRESSION PROGRAM DATA> to be sent by the **device**.

8.7.12.2 Encoding Syntax

An <EXPRESSION RESPONSE DATA> element is defined as



where

<response expression> is defined as:

1) A sequence of AsCII-encoded data bytes in the range 20 to 7E hexadecimal (32 to 126 decimal) except the double quote, number sign, lowercase alpha, left parenthesis, right parenthesis, and semicolon (characters 22, 23, 28, 29, 3B, and 61-7A hexadecimal or 34, 35, 40, 41, 59, and 97-122 decimal respectively), and/or
2) A device-defined set of the <RESPONSE DATA> elements described in 8.7 except <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> and <ARBITRARY ASCII RESPONSE DATA>. This allows the representation of nested expressions.

8.7.12.3 Rules

The <response expression> semantics are completely device-dependent.

A **device** may send properly nested subexpressions within a <response expression>. A subexpression is a syntactically correct <response expression> that is completely contained within a <response expression>.

The **device** documentation shall clearly indicate which <RESPONSE DATA> elements may appear within a <response expression> as well as the maximum subexpression nesting depth. The **device** documentation shall clearly indicate any additional syntax restrictions that the **device** may place on the <response expression>.

The characters listed in 8.7.12.2 (double quote, number sign, lowercase alpha, left and right parentheses, and semicolon) may not appear within the body of the <response expression> but may appear within a <STRING RESPONSE DATA> or <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element in a <response expression> or subexpression. In addition, left and right parentheses may appear in a <response expression> as part of an included subexpression that includes one or more <EXPRESSION PROGRAM DATA> elements.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

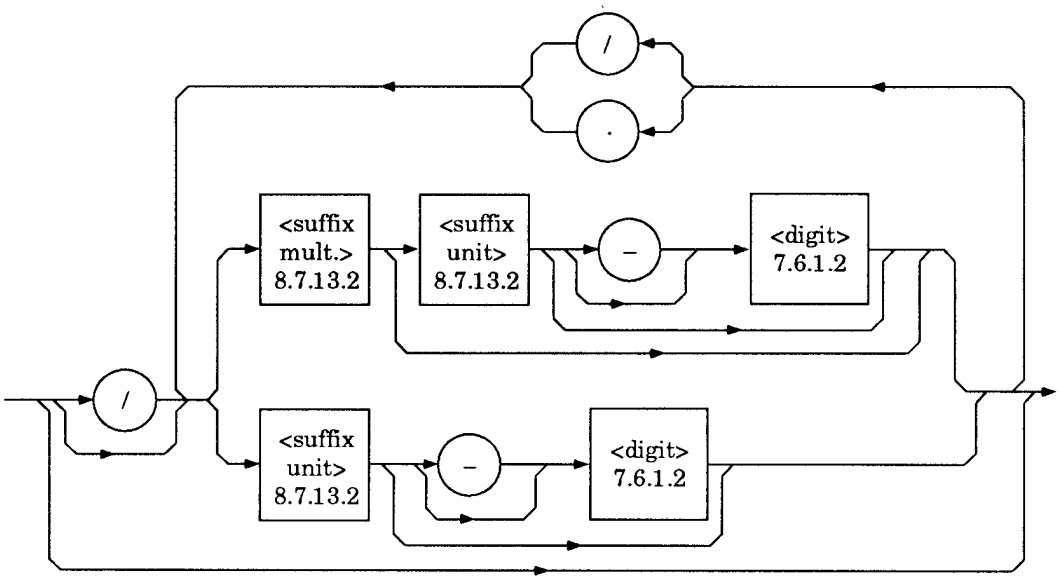
8.7.13 <SUFFIX RESPONSE DATA>

8.7.13.1 Function

A <SUFFIX RESPONSE DATA> element is used to express a unit of operation and a multiplier, if present.

8.7.13.2 Encoding Syntax

A <SUFFIX RESPONSE DATA> is defined as



where

1) <digit> is defined in 7.6.1.2
2) <suffix mult.>, suffix multiplier, is defined in 7.7.3.4.3 as a sequence of one, two, or three <uppercase alpha> characters.
3) <suffix unit> is defined in 7.7.3.4.1 as a sequence of one to four <uppercase alpha> characters. <uppercase alpha> is defined in 8.6.2.

8.7.13.3 Rules

If a **device** uses suffixes, it should make use of the <suffix mult.> and <suffix unit> elements defined in 7.7.3.4.3 and 7.7.3.4.1, respectively. The contents of a <SUFFIX RESPONSE DATA> element shall be either

1) The same as any <SUFFIX RESPONSE DATA> element the **device** can accept.
2) The default units for a <DECIMAL NUMERIC DATA> element with all the alpha characters being uppercase. This element shall not be considered as an extension to the <CHARACTER RESPONSE DATA>.

9. Message Data Coding

This standard specifies specific forms of coding for device-specific messages. The ASCII 7 bit code is the code used throughout this document for the majority of the syntactic and semantic definitions. 8 bit coding is allowed in special

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

block message elements: <ARBITRARY BLOCK PROGRAM DATA>, see 7.7.6, <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>, see 8.7.9, and <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>, see 8.7.10.

9.1 ASCII 7 bit Codes

ANSI X3.4-1986 [1] ASCII 7 bit code is the common data representation code for communication of device-specific messages as described throughout Sections 7. and 8.

The ASCII 7 bit code bits shall be assigned to the DIO signal lines as shown in Table 9-1. The ASCII 7 bit code chart is shown in Table 9-2.

Table 9-1—ASCII 7 Bit Code/Line Relationship

ASCII Code:	*	B7	B6	B5	B4	B3	B2	B1
DIO Line:	DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1

*Bit 8 (DIO8) shall be sent FALSE and ignored on reception by **devices**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,`,,,-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 9-2—ASCII 7 Bit Code Chart

B7 B6 B5 / BITS B4 B3 B2 B1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		NUMBERS SYMBOLS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0	20 DLE 16	40 SP 32	60 0 48	100 @ 64	120 P 80	140 ` 96	160 p 112
0 0 0 1	1 GTL SOH 1	21 LLO DC1 17	41 ! 33	61 1 49	101 A 65	121 Q 81	141 a 97	161 q 113
0 0 1 0	2 STX 2	22 DC2 18	42 " 34	62 2 50	102 B 66	122 R 82	142 b 98	162 r 114
0 0 1 1	3 ETX 3	23 DC3 19	43 # 35	63 3 51	103 C 67	123 S 83	143 c 99	163 s 115
0 1 0 0	4 SDC EOT 4	24 DCL DC4 20	44 & 36	64 4 52	104 D 68	124 T 84	144 d 100	164 t 116
0 1 0 1	5 PPC ENQ 5	25 PPU NAK 21	45 % 37	65 5 53	105 E 69	125 U 85	145 e 101	165 u 117
0 1 1 0	6 ACK 6	26 SYN 22	46 & 38	66 6 54	106 F 70	126 V 86	146 f 102	166 v 118
0 1 1 1	7 BEL 7	27 ETB 23	47 ' 39	67 7 55	107 G 71	127 W 87	147 g 103	167 w 119
1 0 0 0	10 GET BS 8	30 SPE CAN 24	50 (40	70 8 56	110 H 72	130 X 88	150 h 104	170 x 120
1 0 0 1	11 TCT HT 9	31 SPD EM 25	51) 41	71 9 57	111 I 73	131 Y 89	151 i 105	171 y 121
1 0 1 0	12 LF A 10	32 SUB 1A 26	52 * 42	72 : 58	112 J 74	132 Z 90	152 j 106	172 z 122
1 0 1 1	13 VT B 11	33 ESC 1B 27	53 + 43	73 ; 59	113 K 75	133 [91	153 k 107	173 { 123

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Table 9-2 — ASCII 7 Bit Code Chart (continued)

				ADRS CMDS	UNIV CMDS	LISTEN ADDRESSES				TALK ADDRESSES				SECONDARY ADDRESSES OR COMMANDS			
1	1	0	0	14 FF C 12	34 FS 1C 28	54 , 2C 44	12	74 < 3C 60	28	114 L 4C 76	12	134 \ 5C 92	28	154 l 6C 108	12	174 | 7C 124	28
1	1	0	1	15 CR D 13	35 GS 1D 29	55 - 2D 45	13	75 = 3D 61	29	115 M 4D 77	13	135] 5D 93	29	155 m 6D 109	13	175 } 7D 125	29
1	1	1	0	16 SO E 14	36 RS 1E 30	56 . 2E 46	14	76 > 3E 62	30	116 N 4E 78	14	136 ^ 5E 94	30	156 n 6E 110	14	176 ~ 7E 126	30
1	1	1	1	17 SI F 15	37 US 1F 31	57 / 2F 47	15	77 ? 3F 63	UNL	117 O 4F 79	15	137 _ 5F 95	UNT	157 o 6F 111	15	177 DEL (RUBOUT) 7F 127	

KEY

octal 25 PPU
NAK
hex 15 21

IEEE Std 488.1 code
ASCII character
decimal

Copyright (C) 1978, 1980, 1987 Tektronix, Inc. Used with permission.

The dashed area corresponds to the range of codes defined as <white space>.

9.2 Binary 8 Bit Integer Codes

The following codes are the recommended formats for use with block program and response elements, see 7.7.6, 8.7.9, and 8.7.10. Although other codes are not prohibited by this standard, the use of the following codes is preferred.

9.2.1 Byte Order and Data Line Relationships

Bit binary codes utilize high and low bits to directly represent binary (radix 2) numbers. Each binary number is sent in a data field consisting of as many 8 bit bytes as desired. The contents of the data field shall be sent most significant byte first.

A typical data field is illustrated below.

Sequence:	| <-byte 1-> |	| <-byte 2-> |
	8 7 6 5 4 3 2 1 ^	8 7 6 5 4 3 2 1 ^
	MSB (most significant bit position	LSB (least significant bit position)

Binary codes shall be 1, 2, 4, or 8 bytes wide.

9.2.2 Binary Integer Code

Binary integer is assumed to be right-justified in the data field, with the radix point to the right of the least significant bit position. “2”s complement representation is assumed and the contents of the most significant bit position is

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

interpreted as the sign bit. If the **device** data word is not as wide as the field, then the most significant (sign) bit shall be extended to fill the field.

9.2.3 Binary Unsigned Integer Code

Binary unsigned integer is right-justified as described earlier. The contents of the data field's most significant bit is not interpreted as a sign, but as the most significant bit of a positive number. Thus, any most-significant unused data bits in the data field should be zero filled.

9.3 Binary Floating Point Code

The floating point representation included here is a subset of IEEE Std 754-1985 [5].

9.3.1 Floating Point Code Fields

Floating point numbers shall be represented by three fields. The fields are

1) Sign field
2) Exponent field
3) Fraction field

The size of the fields depends on the precision of the number.

For single precision numbers,

Sign field width	1 bit	E(max)	+127
Exponent field width	8 bits	E(min)	−126
Fraction field width	23 bits	Exponent bias	+127
Total width	32 bits		

For double precision numbers,

Sign field width	1 bit	E(max)	+1023
Exponent field width	11 bits	E(min)	−1022
Fraction field width	52 bits	Exponent bias	+1023
Total width	64 bits		

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

9.3.2 Basic Formats

Numbers in the single and double formats are composed of the following three fields:

(1) 1 bit sign	s (s = 0 = positive) (s = 1 = negative)
(2) Biased exponent	e = E + bias
(3) Fraction	f = .b(1)b(2) ... b(p – 1)

where

p = the number of significant bits
b(n) = 0 or 1

The range of the unbiased exponent, E, shall include every integer between two values E(min) and E(max), inclusive, and also two other reserved values E(min) – 1 to encode ±0 and denormalized numbers, and E(max)+l to encode ±∞ and NaNs (Not a Number symbol). The foregoing parameters are given in 9.3.1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows:

9.3.2.1 Single

A 32 bit single format number, X, is divided as shown in 9.3.2. The value, *v*, of X is inferred from its constituent fields, thus

1) If e = 255 and f ≠ 0, then *v* is NaNs regardless of s
2) If e = 255 and f = 0, then $v = (-1)^S \infty$
3) If 0 < e < 255, then $v = (-1)^S 2^{e-127}(1.f)$
4) If e = 0 and f ≠ 0, then $v = (-1)^S 2^{-126}(0.f)$
   (denormalized numbers)
5) If e = 0 and f = 0, then $v = (-1)^S 0$ (zero)

9.3.2.2 Double

A 64 bit double format number, X, is divided as shown in 9.3.2. The value, *v*, of X is inferred from its constituent fields thus

1) If e = 2047 and f ≠ 0, then *v* is NaNs regardless of s
2) If e = 2047 and f = 0, then $v = (-1)^S \infty$
3) If 0 < e < 2047, then $v = (-1)^S 2^{e-1023} (1.f)$
4) If e = 0 and f ≠ 0, then $v = (-1)^S 2^{-1022} (0.f)$
   (denormalized numbers)
5) If e = 0 and f = 0, then $v = (-1)^S 0$ (zero)

9.3.3 Order of Transmission

9.3.3.1 Single Precision Numbers

Single precision numbers shall be transmitted in 4 bytes. The transmission shall be structured according to the following relationships between DIO signal lines and the fields.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

DIO	—	8	7	6	5	4	3	2	1	
		S	E ^	E	E	E	E	E	E	First byte sent
		MSBE								
		E ^	F ^	F	F	F	F	F	F	Second byte sent
		LSBE	MSBF							
		F	F	F	F	F	F	F	F	Third byte sent
		F	F	F	F	F	F	F	F ^	Fourth byte sent
								LSBF		

where

MSBE	is the most significant bit of the exponent
LSBE	is the least significant bit of the exponent
MSBF	is the most significant bit of the fraction
LSBF	is the least significant bit of the fraction
S	is the sign bit
E	is an exponent bit
F	is a fraction bit

9.3.3.2 Double Precision Numbers

Double precision numbers shall be transmitted in 8. bytes. The transmission shall be structured according to the following relationships between DIO signal lines and the fields.

DIO	—	8	7	6	5	4	3	2	1	
		S	E ^	E	E	E	E	E	E	First byte sent
		MSBE								
		E	E	E	E ^	F ^	F	F	F	Second byte sent
					LSBE	MSBF				
		F	F	F	F	F	F	F	F	Third through seventh byte sent
		F	F	F	F	F	F	F	F ^	Eighth byte sent
								LSBF		

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

where

MSBE	is the most significant bit of the exponent
LSBE	is the least significant bit of the exponent
MSBF	is the most significant bit of the fraction
LSBF	is the least significant bit of the fraction
S	is the sign bit
E	is an exponent bit
F	is a fraction bit

9.3.4 Example of Single Precision Number

An example of a single precision number could be encoded using these four bytes:

01000010	11100000	00000000	00000000
se———	ef———	———	———f

where

	binary	**decimal**
s =	0	= 0
e =	10000101	= 133
f =	.110	=.75

The number then evaluates to:

$v = (-1)^S 2^{e-127}(1 .f)$
$= (-1)^0 2^6 (1.75)$
$= (64)(1.75)$
$= 112$

10. Common Commands and Queries

This section describes common commands and queries. Descriptive information and compliance requirements are included for each command. Table 9-2 lists alphabetically all the common command and query headers defined in this standard. Table 10-1 lists the same program messages grouped according to function. Syntax of the commands and queries follows the conventions of 7.1. Common commands and queries, with the exception of *AAD, see 10.1, and *DLF, see 10.2, are described as <PROGRAM MESSAGE UNIT> elements, see Fig 7-3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 10-1—IEEE 488.2 Common Command Headers

Mnemonic	Name	Section
*AAD	Accept Address Command	10.1
*CAL?	Calibration Query	10.2
*CLS	Clear Status Command	10.3
*DDT	Define Device Trigger Command	10.4
*DDT?	Define Device Trigger Query	10.5
*DLF	Disable Listener Function Command	10.6
*DMC	Define Macro Command	10.7
*EMC	Enable Macro Command	10.8
*EMC?	Enable Macro Query	10.9
*ESE	Standard Event Status Enable Command	10.10
*ESE?	Standard Event Status Enable Query	10.11
*ESR?	Standard Event Status Register Query	10.12
*GMC?	Get Macro Contents Query	10.13
*IDN?	Identification Query	10.14
*IST?	Individual Status Query	10.15
*LMC?	Learn Macro Query	10.16
*LRN?	Learn Device Setup Query	10.17
*OPC	Operation Complete Command	10.18
*OPC?	Operation Complete Query	10.19
*OPT?	Option Identification Query	10.20
*PCB	Pass. Control Back Command	10.21
*PMC	Purge Macros Command	10.22
*PRE	Parallel Poll Enable Register Command	10.23
*PRE?	Parallel Poll Enable Register Query	10.24
*PSC	Power-On Status Clear Command	10.25
*PSC?	Power-On Status Clear Query	10.26
*PUD	Protected User Data Command	10.27
*PUD?	Protected User Data Query	10.28
*RCL	Recall Command	10.29
*RDT	Resource Description Transfer Command	10.30
*RDT?	Resource Description Transfer Query	10.31
*RMC	Remove Individual Macro Command	10.40
*RST	Reset Command	10.32
*SAV	Save Command	10.33
*SDS	Save Default Device Setting Command	10.41
*SRE	Service Request Enable Command	10.34
*SRE?	Service Request Enable Query	10.35
*STB?	Read Status Byte Query	10.36
*TRG	Trigger Command	10.37
*TST?	Self-Test Query	10.38
*WAI	Wait-to-Continue Command	10.39

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 10-2—IEEE 488.2 Common Command Groups

Mnemonic	Group	Compliance	Section
*AAD	Auto Configure	Optional*	10.1
*DLF	Auto Configure	Optional	10.6
*IDN?	System Data	Mandatory	10.14
*OPT?	System Data	Optional	10.20
*PUD	System Data	Optional	10.27
*PUD?	System Data	Optional	10.28
*RDT	System Data	Optional	10.30
*RDT?	System Data	Optional	10.31
*CAL?	Internal Operations	Optional	10.2
*LRN?	Internal Operations	Optional	10.17
*RST	Internal Operations	Mandatory	10.32
*TST?	Internal Operations	Mandatory	10.38
*OPC	Synchronization	Mandatory	10.18
*OPC?	Synchronization	Mandatory	10.19
*WAI	Synchronization	Mandatory	10.39
*DMC	Macro	Optional	10.7
*EMC	Macro	Optional	10.8
*EMC?	Macro	Optional	10.9
*GMC?	Macro	Optional	10.13
*LMC?	Macro	Optional	10.16
*PMC	Macro	Optional	10.22
*RMC	Macro Extended	Optional†	10.40
*IST?	Parallel Poll	Mandatory if PP1	10.15
*PRE	Parallel Poll	Mandator if PP1	10.23
*PRE?	Parallel Poll	Mandatory if PP1	10.24
*CLS	Status & Event	Mandatory	10.3
*ESE	Status & Event	Mandatory	10.10
*ESE?	Status & Event	Mandatory	10.11
*ESR?	Status & Event	Mandatory	10.12
*PSC	Status & Event	Optional	10.25
*PSC?	Status & Event	Optional	10.26
*SRE	Status & Event	Mandatory	10.34
*SRE?	Status & Event	Mandatory	10.35
*STB?	Status & Event	Mandatory	10.36
*DDT	Trigger	Optional; requires DT1	10.4
*DDT?	Trigger	Optional; requires DT1	10.5
*TRG	Trigger	Mandatory if DT1	10.37
*PCB	Controller	Mandatory if other than C0	10.21
*RCL	Stored Settings	Optional	10.29
*SAV	Stored Settings	Optional	10.33
*SDS	Stored Setting Extended	Optional‡	10.41

*If any commands in either the Auto Configure, Macro, or Stored Settings groups are implemented, then all the commands in that group shall be implemented.
†Implementation of this command requires implementation of the Macro group commands.
‡Implementation of this command requires implementation of the Stored Settings group commands.

Section 6.1.6.1.1 requires that a **device** shall generate a Command Error if an unimplemented common command is received.

--`,,`,,,-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.1 *AAD, Accept Address Command

10.1.1 Function and Requirements

This command, in conjunction with the Address Set protocol, allows the **controller** to detect all address-configurable **devices** (that is, **devices** that implement this command) and assign an IEEE 488.1 address to each of those **devices**. An address-configurable **device** is detected when the **controller** has completed a byte-by-byte search of the **device's** identifier, after which time the address is assigned. The Address Set protocol causes this identifier search to be executed repeatedly until all address-configurable **devices** have been detected. See 13.4.2 for detailed implementation requirements of this command.

10.1.2 Syntax

The syntax for the Accept Address command is defined as a <COMMAND PROGRAM HEADER> followed immediately by a <PROGRAM MESSAGE TERMINATOR>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as " *AAD."

10.1.3 Semantics

See 13.4.2.

10.1.4 Related Common Commands

***DLF** — Implementation requires this optional command.

10.1.5 Standard Compliance

Optional.

10.1.6 Error Handling

See 13.4.2.

10.2 *CAL?, Calibration Query

10.2.1 Function and Requirements

The Calibration query causes a **device** to perform an internal self-calibration and generate a response that indicates whether or not the **device** completed the self-calibration without error. Additional information about any calibration errors may be contained in the response, see 10.2.3.

The Calibration query shall not require any local operator interaction to function. It shall not create bus conditions that are violations to the IEEE Std 488.1-1987 [4] or IEEE Std 488.2-1992 standards. Otherwise, the scope of the self-calibration is completely at the discretion of the **device** designer.

Upon completion of *CAL?, the **device** should be returned to the state just prior to the calibration cycle, or other specified operational condition stated in the documentation, see 4.9.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.2.2 Query Structure

10.2.2.1 Query Syntax

The syntax for the Calibration query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*CAL?."

10.2.2.2 Response Syntax

The response syntax for the Calibration query is defined as a single <NR1 NUMERIC RESPONSE DATA>. The <NR1 NUMERIC RESPONSE DATA> shall be in the range of –32767 to 32767.

10.2.3

A value of zero indicates the calibration completed without any detected errors. A value not equal to zero indicates the calibration did not complete or completed with errors detected. The semantics of the nonzero response shall be completely at the discretion of the **device** designer.

10.2.4 Related Common Commands

None.

10.2.5 Standard Compliance

Optional, requires no other optional commands.

10.3 *CLS, Clear Status Command

10.3.1 Function and Requirements

The Clear Status command clears status data structures, see 11.1.2, and forces the **device** to the Operation Complete Command Idle State and the Operation Complete Query Idle State, see 12.5.2 and 12.5.3.

If the Clear Status command immediately follows a <PROGRAM MESSAGE TERMINATOR>, the Output Queue and the MAV bit will be cleared because any new <PROGRAM MESSAGE> after a <PROGRAM MESSAGE TERMINATOR> clears the Output Queue, see 6.3.2.3.

See Appendix D. for a discussion of the resetting actions of *CLS as they relate to other reset commands.

10.3.2 Syntax

The syntax for the Clear Status command is defined as only a <COMMAND PROGRAM HEADER>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*CLS."

10.3.3 Semantics

Not applicable.

10.3.4 Related Common Commands

None.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.3.5 Standard Compliance

Mandatory.

10.4 *DDT, Define Device Trigger Command

10.4.1 Function and Requirements

The Define Device Trigger command stores a command sequence that is executed when a group execute trigger (GET), IEEE 488.1 interface message, or *TRG common command is received, see 10.37. The *RST common command shall set the command sequence to a device-defined state, see 10.32.

If a **device** implements *DDT, there shall exist a command sequence to represent any GET or *TRG action. The **device** documentation shall state the command sequence sent with the *DDT command that has the same effect as *RST on the action performed by GET.*TRG may be used for this purpose, see 10.4.6.3.

10.4.2 Syntax

There are two syntax formats for the Define Device Trigger command:

1) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by an <ARBITRARY BLOCK PROGRAM DATA> element
2) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <STRING PROGRAM DATA> element

Devices that implement *DDT shall accept format (1) above, and may additionally accept format (2).

The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*DDT."

10.4.3 Semantics

The <ARBITRARY BLOCK PROGRAM DATA> or <STRING PROGRAM DATA> contains either the sequence of <PROGRAM MESSAGE UNIT> elements to be executed or a zero-length data field. A zero-length field indicates that no action will be taken when a group execute trigger (GET) IEEE 488.1 interface message or a *TRG common command, see 10.37, is received.

NOTE — The alternate <STRING PROGRAM DATA> format may only contain 7 bit ASCII data bytes, thus restricting its usage.

10.4.4 Related Common Commands

*DDT? — This is the companion query.

10.4.5 Standard Compliance

Optional, but requires the DT1 subset.

10.4.6 Error Handling

10.4.6.1 Oversized Command Sequence

An Execution Error, see 6.1.7 and 11.5.1.1.5, shall be reported if the block or string is too long for the **device** to accept. The maximum length requirement shall be specified in the **device's** documentation.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.4.6.2 Command Sequence Errors

Any Command or Execution Errors contained in the command sequence may be either detected immediately or deferred until the macro is executed.

10.4.6.3 Sequence Containing *TRG

If the stored command sequence contains *TRG or the label of an enabled macro whose expansion leads to the execution of *TRG, then the **device** may respond in one of the following two ways:

1) The **device** may consider this condition as being recursion. The application program can only stop this recursion by sending DCL or SDC. The **device** may either allow recursion to take place indefinitely or report an Execution Error at a device-specified level of recursion.
2) The **device** may interpret the *TRG as meaning the **device** should perform that default action for *TRG, i.e., the action assigned to *TRG by *RST. Thus, a sequence containing only *TRG sets the definition of *TRG to the *RST state. When *TRG is included with other commands or queries in the sequence, the **device** executes the commands, queries, and default *TRG action in the order specified in the sequence.

Which method is chosen and the manner in which recursion is handled shall be stated in the **device** documentation. See 4.9.

10.4.7 Example

*DDT #217TRIG WFM; MEASWFM?

A GET will immediately trigger a waveform and perform a query that places the acquired waveform into the **device's** Output Queue.

10.5 *DDT?, Define Device Trigger Query

10.5.1 Function and Requirements

The Define Device Trigger query allows the programmer to examine the command sequence which will be executed when a GET or *TRG command is received.

10.5.2 Query Structure

10.5.2.1 Query Syntax

The syntax for the Define Device Trigger query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*DDT?."

10.5.2.2 Response Syntax

The response syntax for the Define Device Trigger query is defined as a <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element.

10.5.3 Response Semantics

The <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element contains the sequence of <PROGRAM MESSAGE UNIT> elements that are executed when a GET or *TRG is received.

The condition of "no command sequence stored" results in a response of a zero-length block.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.5.4 Related Common Commands

*DDT — Implementation requires this optional command. This is the companion command.

10.5.5 Standard Compliance

Optional, but requires the DT1 subset.

10.6 *DLF, Disable Listener Function Command

10.6.1 Function and Requirements

The Disable Listener Function command causes a **device** to cease being a listener (change to L0 subset). If this command is the first device-specific message received after the **device** leaves IEEE 488.1 DCAS state, the **device** shall cease being a listener within 100 ms after the acceptance of the <PROGRAM MESSAGE TERMINATOR>. See 13.4.1 for detailed implementation requirements of this command. A subsequent DCL message shall restore listener capability. The **device** shall resume listening within 100 ms after entering IEEE 488.1 DCAS state.

10.6.2 Syntax

The syntax for the Disable Listener Function command is defined as a <COMMAND PROGRAM HEADER> followed immediately by a <PROGRAM MESSAGE TERMINATOR>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*DLF."

10.6.3 Semantics

The *DLF command shall be immediately followed by a <PROGRAM MESSAGE TERMINATOR> with no parameters.

10.6.4 Related Common Commands

*AAD — Implementation requires this optional command.

10.6.5 Standard Compliance

Optional.

10.6.6 Error Handling

See 13.4.1 and 13.4.2 for further information.

10.7 *DMC, Define Macro Command

10.7.1 Function and Requirements

The Define Macro command allows the programmer to assign a sequence of zero or more <PROGRAM MESSAGE UNIT> elements to a macro label. The sequence is executed when the label is received as a <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER>.

10.7.2 Syntax

There are two syntax formats for the Define Macro command.

1) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <STRING PROGRAM DATA> followed by a <PROGRAM DATA SEPARATOR> followed by an <ARBITRARY BLOCK PROGRAM DATA> element.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

2) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <STRING PROGRAM DATA> followed by a <PROGRAM DATA SEPARATOR> followed by <STRING PROGRAM DATA> element.

Devices that implement *DMC shall accept format (1) above, and may additionally accept format (2).

The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*DMC."

10.7.3 Semantics

The first <STRING PROGRAM DATA> element contains the macro label. The macro label shall be either a <simple command program header>, a <compound command program header>, a <simple query program header>, or a <compound query program header>. The macro label shall not, however, be either a <common command program header>, see 7.6.1.2, or a <common query program header>, see 7.6.2.2. Any <white space> or a leading colon (if compound headers are implemented), or both, appearing before the first character of a macro label shall not be considered a part of that label.

The macro label may be the same as a device-specific <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER>. In this case, provided macros are enabled, see 10.8.4, and the macro label received within a <PROGRAM MESSAGE> shall be executed as the macro expansion, not as the device-specific command of the same name. That device-specific command may still be executed by disabling macros.

The <ARBITRARY BLOCK PROGRAM DATA> or second <STRING PROGRAM DATA> element contains the sequence of <PROGRAM MESSAGE UNIT> elements being labeled.

NOTE — The alternate <STRING PROGRAM DATA> format may only contain 7 bit ASCII data bytes, thus restricting its usage.

Parameters may be passed to the sequence during execution. Placeholders for parameters appear in the sequence as a dollar sign (ASCII codes 31-39, 49-57 decimal). The first <PROGRAM DATA> element following the macro label, when used as a header, is substituted for the parameter placeholder labeled $1. The second <PROGRAM DATA> element is substituted for $2. This substitution process is continued for up to nine parameters.

The placeholder must appear in the sequence as if it were a complete <PROGRAM DATA> element. See 7.3.2 for a description of where <PROGRAM DATA> elements are permitted in a <PROGRAM MESSAGE UNIT>. Paragraph 7.7.7 describes how <PROGRAM DATA> elements may appear in an <expression>.

Devices that accept <compound command program header> and <compound query program header> elements should prefix the header-path before comparing the macro label against the stored macro names. The match shall be exact except for the leading colon.

NOTE — Since different paths to the same :node may exist, all possible paths shall be considered when doing the comparison.

The header-path is formed in the same manner that is used when <compound command program header> and <compound query program header> elements that are not macros are received.

NOTE — A macro label with an appended question mark should not imply any action other than the appropriate error response, unless a separate *DMC command explicitly defining the query was received.

10.7.4 Related Common Commands

*PMC — Implementation requires this optional command.

*GMC? — Implementation requires this optional query.

*LMC — Implementation requires this optional command.

*EMC — Implementation requires this optional command.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

*EMC? — Implementation requires this optional query.

*RMC — Implementation optional.

10.7.5 Standard Compliance

Optional.

10.7.6 Error Handling

10.7.6.1 Oversized Command Sequence

An Execution Error, see 6.1.7 and 11.5.1.1.5, shall be reported if the block or string is too long for the **device** to accept.

The maximum length requirement is device-specific and shall be specified in the **device** documentation.

10.7.6.2 Macro Label Errors

A **device** shall report an Execution Error if the macro label is too long for the **device** to accept.

The maximum label length is device-specific and shall be specified in the **device** documentation.

The **device** shall report an Execution Error if the macro label does not adhere to the <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER> syntax in 7.6.

If a **device** does not implement compound headers, it shall only allow <simple command program header> or <simple query program header> elements as macro labels. **Devices** that implement only simple headers shall report an Execution Error if the macro label contains a colon.

Macro labels that contain too many colon-separated mnemonics shall cause an Execution Error. The level of complexity shall be included in the **device** documentation.

10.7.6.3 Command Sequence Errors

Any Command or Execution Errors contained in the command sequence may be either detected immediately or deferred until the macro is executed.

10.7.6.4 Redefining Existing Macros

Redefining an existing macro shall cause an Execution Error.

10.7.6.5 Parameter Errors

If the command sequence requires a different number of parameters than follow the macro label, the **device** shall generate a Command Error. The **device** shall perform normal parameter checking after parameter substitution.

10.7.6.6 Recursion

If the sequence of <PROGRAM MESSAGE UNIT> elements contains the label of an enabled macro whose expansion leads to execution of the macro being defined, then recursion will occur. The application program can only step this recursion by sending DCL or SDC. The **device** may either allow recursion to take place indefinitely or report an Execution Error at a device-defined level of recursion. The manner in which recursion is handled must be stated in the **device** documentation. See 4.9.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.7.7 Examples

1) *DMC“HOME”, #18MOVE 0,0
The macro descriptively labels a command that sends a pen plotter to its home position.
2) *DMC “SLO”, “SPEED SLOW’
The macro defines an abbreviation of a longer command, thus reducing bus traffic.
3) *DMC “SETUP1”, #221VOLT 14.5;CURLIM 2E-3
SETUP1 will be expanded by the **device** into VOLT 14.5;CURLIM 2E-3
4) *DMC “TEST_A”, “BEGINFREQ $1;ENDFREQ $2”
This example illustrates the use of parameters. This macro expects two parameters. The **device** interprets TEST_A 1000,2000 the same as BEGINFREQ 1000; ENDFREQ 2000.
5) *DMC “CLK_EXT”,#219HORIZ:CLOCK:EXT:NEG
The command that selects an external clock for the horizontal trigger of an oscilloscope with negative edge sensitivity is given a shorthand notation.
6) *DMC “SWEEP_SET”,“START $1;MARK1 $1;STOP $2;MARK2 $2”
This macro demonstrates the reuse of parameters. The first parameter is used to set both the start frequency as well as the position of one of the markers. The second parameter sets the stop frequency and another marker. SWEEP_SET 1E6,5E6 is equivalent to START 1EI6;MARK1 1E6;STOP 5E6;MARK2 5E6.

10.8 *EMC, Enable Macro Command

10.8.1 Function and Requirements

The Enable Macro command enables and disables expansion of macros. Macro definitions are not affected by this command. One use of this command is to turn off macro expansion in order to execute a device-specific command with the same name as a macro. The *RST command disables the expansion of macros, see 10.32.

10.8.2 Syntax

The syntax for the Enable Macro command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as “*EMC.”

10.8.3 Semantics

The value of the <DECIMAL NUMERIC PROGRAM DATA> determines whether the defined macros are enabled or disabled.

A <DECIMAL NUMERIC PROGRAM DATA> that rounds to an integer value of zero disables any defined macros.

A <DECIMAL NUMERIC PROGRAM DATA> that rounds to an integer value not equal to zero enables any defined macros. The allowed range shall be –32767 to +32767.

10.8.4 Related Common Commands

*PMC — Implementation requires this optional command.
*GMC? — Implementation requires this optional query.
*DMC — Implementation requires this optional command.
*LMC — Implementation requires this optional command.
*EMC? — Implementation requires this optional query.
*RMC — Implementation optional.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.8.5 Standard Compliance

Optional.

10.8.6 Error Handling

If the <DECIMAL NUMERIC PROGRAM DATA> element does not round to an integer in the range of-32767 to +32767 inclusive, an Execution Error shall be reported.

10.8.7 Example

Assume an ac power supply has a command of “OUTPUT” that requires two parameters: voltage and frequency. The following new definition redefines “OUTPUT” to take only one parameter, voltage. The frequency is always 1 kHz.

*DMC “OUTPUT”,#228*EMC *;OUTPUT $1,1000;*EMC 1

10.9 *EMC?, Enable Macro Query

10.9.1 Function and Requirements

The Enable Macro query allows the programmer to query whether the macros are enabled. A returned value of zero indicates that macros are disabled. A returned value of one indicates that macros are enabled.

10.9.2 Query Structure

10.9.2.1 Query Syntax

The syntax for the Enable Macro query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as “*EMC?.”

10.9.2.2 Response Syntax

The response syntax for the Enable Macro query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> shall be either a single ASCII-encoded byte for “0” (30, 48 decimal) or “1” (31, 49 decimal).

10.9.3 Response Semantics

A value of zero indicates that macros are disabled. A value of one indicates that macros are enabled.

10.9.4 Related Common Commands

*PMC — Implementation requires this optional command.
*GMC? — Implementation requires this optional query.
*DMC — Implementation requires this optional command.
*LMC — Implementation requires this optional command.
*EMC — Implementation requires this optional command.
*RMC — Implementation optional.

10.9.5 Standard Compliance

Optional.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.10 *ESE, Standard Event Status Enable Command

10.10.1 Function and Requirements

The Standard Event Status Enable command sets the Standard Event Status Enable Register bits as defined in 11.5.1.3.

10.10.2 Syntax

The syntax for the Standard Event Status Enable command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*ESE."

10.10.3 Semantics

The <DECIMAL NUMERIC PROGRAM DATA>, when rounded to an integer value and expressed in base 2 (binary), represents the bit values of the Standard Event Status Enable Register. See 11.4.2.3.

The value of the integer shall be in the range of 0 through 255.

10.10.4 Related Common Commands

*ESE? — This is the companion query.

*PSC — Determines whether the Standard Event Status Enable Register is cleared at power-on.

10.10.5 Standard Compliance

Mandatory.

10.10.6 Error Handling

An out-of-range integer shall cause an Execution Error, see 11.5.1.1.5.

10.11 *ESE?, Standard Event Status Enable Query

10.11.1 Function and Requirements

The Standard Event Status Enable query allows the programmer to determine the current contents of the Standard Event Status Enable Register. See 11.5.1.3.

10.11.2 Query Structure

10.11.2.1 Query Syntax

The syntax for the Standard Event Status Enable query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*ESE?."

10.11.2.2 Response Syntax

The response syntax for the Standard Event Status Enable query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> shall be in the range of 0 through 255.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.11.3 Response Semantics

The <NR1 NUMERIC RESPONSE DATA> integer value expressed in base 2 (binary) represents the bit values of the Standard Event Status Enable register. See 11.4.2.3.2.

10.11.4 Related Common Commands

*ESE —This is the companion command.

10.11.5 Standard Compliance

Mandatory.

10.12 *ESR?, Standard Event Status Register Query

10.12.1 Function and Requirements

The Standard Event Status Register query allows the programmer to determine the current contents of the Standard Event Status Register. Reading the Standard Event Status Register clears it. See 11.5.1.2.

10.12.2 Query Structure

10.12.2.1 Query Syntax

The syntax for the Standard Event Status Register query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*ESR?."

10.12.2.2 Response Syntax

The response syntax for the Standard Event Status Register query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> shall be in the range of 0 through 255.

10.12.3 Response Semantics

The <NR1 NUMERIC RESPONSE DATA> integer value expressed in base 2 (binary) represents the bit values of the Standard Event Status Register. See 11.5.1.

10.12.4 Related Common Commands

None.

10.12.5 Standard Compliance

Mandatory.

10.13 *GMC?, Get Macro Contents Query

10.13.1 Function and Requirements

The Get Macro Contents query allows the current definition of a macro to be retrieved from a **device**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.13.2 Query Structure

10.13.2.1 Query Syntax

The syntax for the Get Macro Contents query is defined as a <QUERY PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <STRING PROGRAM DATA> element. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*GMC?."

10.13.2.2 Response Syntax

The response syntax for the Get Macro Contents query is defined as a <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element.

10.13.3 Semantics

10.13.3.1 Parameter Semantics

The data in the <STRING PROGRAM DATA> element must be a currently defined macro label. Any <white space> or a leading colon (if compound headers are implemented), or both, appearing before the first character of a macro label shall not be considered a part of that label.

10.13.3.2 Response Semantics

The <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element contains the <PROGRAM MESSAGE UNIT> sequence, which is executed when the macro label is received.

The condition of "no command sequence stored" results in a response of a zero-length block.

10.13.4 Related Common Commands

*PMC — Implementation requires this optional command.
*EMC — Implementation requires this optional command.
*EMC? — Implementation requires this optional query.
*DMC — Implementation requires this optional command.
*LMC — Implementation requires this optional command.
*RMC — Implementation optional.

10.13.5 Standard Compliance

Optional.

10.13.6 Error Handling

If the programmer attempts to retrieve the contents of an undefined macro, the **device** shall return a zero-length block and report an Execution Error.

10.14 *IDN?., Identification Query

10.14.1 Function and Requirements

The intent of the Identification query is for the unique identification of **devices** over the **system interface**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.14.2 Query Structure

10.14.2.1 Query Syntax

The syntax for the Identification query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*IDN?."

10.14.2.2 Response Syntax

The response syntax for the Identification query is defined as an <ARBITRARY ASCII RESPONSE DATA> element. This implies that the *IDN? query should be the last <QUERY MESSAGE UNIT> in a <TERMINATED PROGRAM MESSAGE>, see 6.5.7.5 and 8.7.11.

10.14.3 Response Semantics

The response is organized into four fields separated by commas. The field definitions are as follows:

Field 1	Manufacturer	required
Field 2	Model	required
Field 3	Serial number	ASCII character 0 if not available
Field 4	Firmware level or equivalent	ASCII character 0 if not available

NOTE — ASCII character "0" represents a single ASCII-encoded byte with value 30 (48 decimal).

The presence of data in all the fields is mandatory. If either field 3 or 4 is not available, the ASCII character "0" shall be returned for that field.

A field may contain any 7-bit ASCII-encoded bytes in the range of 20 through 7E (32 through 126 decimal) except commas (2C, 44 decimal) and semicolons (3B, 59 decimal).

The overall length of the *IDN? response shall be less than or equal to 72 characters.

The precise format of fields (except ASCII character 0 as meaning empty field) is left to the **device** designer.

If unique serial numbers are used, then ASCII character 0 shall not be used as a serial number.

10.14.4 Related Common Commands

None.

10.14.5 Standard Compliance

Mandatory.

10.14.6 Example

A hypothetical *IDN? response might appear as follows for a model 246B product from a company, XYZ Co, that has chosen the mnemonic "XYZCO" to represent itself with serial number S000-123-02, and no firmware revision specified.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

XYZCO,246B,S000-0123-02,0

10.15 *IST?, Individual Status Query

10.15.1 Function and Requirements

The Individual Status query allows the programmer to read the current state of the IEEE 488.1 defined “ist” local message in the **device**, see 11.6.2.

10.15.2 Query Structure

10.15.2.1 Query Syntax

The syntax for the Individual Status query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as “*IST?.”

10.15.2.2 Response Syntax

The response syntax for the Individual Status query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> shall be a single ASCII-encoded byte encoded as a “0” or “1” (30 or 31, 48 or 49 decimal).

10.15.3 Response Semantics

A <NR1 NUMERIC RESPONSE DATA> element with the value zero indicates the ist local message is FALSE.

A <NR1 NUMERIC RESPONSE DATA> element with the value of one indicates that the ist local message is TRUE.

The response of this query is dependent upon the current status of the instrument.

10.15.4 Related Common Commands

None.

10.15.5 Standard Compliance

Mandatory for **devices** implementing the PP1 subset.

10.16 *LMC?, Learn Macro Query

10.16.1 Function and Requirements

This query returns the currently defined macro labels.

10.16.2 Query Structure

10.16.2.1 Query Syntax

The syntax for the Learn Macro query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as “*LMC?.”

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.16.2.2 Response Syntax

The response syntax for the Learn Macro query is defined as a sequence of one or more <STRING RESPONSE DATA> elements each separated by a <RESPONSE DATA SEPARATOR>.

10.16.3 Response Semantics

The <STRING RESPONSE DATA> elements contain macro labels currently defined in the **device**. See 10.7.

If no macro labels are defined, then the response shall be a single <STRING RESPONSE DATA> with null string data, that is, two consecutive double quote (3) marks.

The response is independent of the macro-enabled or macro-disabled state of the **device.**

10.16.4 Related Common Commands

See 10.22.

*PMC — Implementation requires this optional command.
*DMC — Implementation requires this optional command.
*EMC — Implementation requires this optional command.
*EMC? — Implementation requires this optional query.
*GMC? — Implementation requires this optional query.
*RMC — Implementation optional.

10.16.5 Standard Compliance

Optional.

10.17 *LRN?, Learn Device Setup Query

10.17.1 Function and Requirements

The Learn Device Setup query allows the programmer to obtain a sequence of<RESPONSE MESSAGE UNIT> elements that may later be used as <PROGRAM MESSAGE UNIT> elements to place the **device** in the state it was in when the *LRN? common query was made.

10.17.2 Query Structure

10.17.2.1 Query Syntax

The syntax for the Learn Device Setup query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*LRN?."

10.17.2.2 Response Syntax

The response syntax for the Learn Device Setup query is defined as a sequence of one or more <RESPONSE MESSAGE UNIT> elements separated by <RESPONSE MESSAGE UNIT SEPARATOR> elements. The <RESPONSE MESSAGE UNIT> shall use the path, shown in Fig 8-3, that contains the <RESPONSE HEADER>.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.17.3 Response Semantics

The <RESPONSE MESSAGE> is a sequence of one or more <RESPONSE MESSAGE UNIT> elements that are directly resendable as <PROGRAM MESSAGE UNIT> elements.

The **device** shall be restored to its "learned" state from any other current state (including the state after *RST) upon receipt of this <RESPONSE MESSAGE> as a <PROGRAM MESSAGE>.

The scope of the restored settings is the same as the optional *SAV command, see 10.33.1.

NOTE — If<learn string> is the response to *LRN?, then receiving <learn string> is equivalent to receiving *RST;<learn string>. The **device** designer shall :implement the necessary <PROGRAM MESSAGE UNIT> elements needed to cause this action.

10.17.4 Related Common Commands

None.

10.17.5 Standard Compliance

Optional.

10.18 *OPC, Operation Complete Command

10.18.1 Function and Requirements

The Operation Complete command causes the **device** to generate the operation complete message in the Standard Event Status Register when all pending selected **device** operations have been finished. See 12.5.2.2 for details of operation.

10.18.2 Syntax

The syntax for the Operation Complete command is defined as only a <COMMAND PROGRAM HEADER>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*OPC."

10.18.3 Semantics

Not applicable.

10.18.4 Related Common Commands

*OPC?, *WAI.

10.18.5 Standard Compliance

Mandatory.

10.19 *OPC?, Operation Complete Query

10.19.1 Function and Requirements

The Operation Complete query places an ASCII character "1" into the **device's** Output Queue when all pending selected **device** operations have been finished. See 12.5.3 for details of operation.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.19.2 Query Structure

10.19.2.1 Query Syntax

The syntax for the Operation Complete query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*OPC?."

10.19.2.2 Response Syntax

The response syntax for the Operation Complete query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> shall be a single ASCII-encoded byte for "1" (31, 49 decimal).

10.19.3 Response Semantics

None.

10.19.4 Related Common Commands

*OPC, *WAI.

10.19.5 Standard Compliance

Mandatory.

10.20 *OPT?, Option Identification Query

10.20.1 Function and Requirements

The Option Identification query is for identifying reportable **device** options over the **system interface.**

10.20.2 Query Structure

10.20.2.1 Query Syntax

The syntax for the Option Identification query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*OPT?."

10.20.2.2 Response Syntax

The response syntax for the Option Identification Query is defined as an <ARBITRARY ASCII RESPONSE DATA> element. This implies that the *OPT? query should be the last <QUERY MESSAGE UNIT> in a <TERMINATED PROGRAM MESSAGE>, see 6.5.7.5 and 8.7.11.

10.20.3 Response Semantics

The response consists of any number of fields separated by commas.

The fields shall follow these rules:

1) Each field shall correspond to a reportable option present in the **device**. The precise format of the fields is left to the **device** designer.
2) Empty fields are prohibited. Missing reportable options respond with an ASCII character "0" (30, 48 decimal).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

3) Option data may be reported in a position-dependent manner.
4) A single ASCII character “0” shall be returned as a response if the **device** contains no reportable options. ASCII character “0” may not be used as an option identification.
5) A field may contain any 7 bit ASCII-encoded bytes in the range of 20 through 7E (32 through 126 decimal) except commas (2C, 44 decimal) and semicolons (3B, 59 decimal).

The overall length of the *OPT? response shall be less than or equal to 255 characters.

10.20.4 Related Common Commands

None.

10.20.5 Standard Compliance

Optional.

10.21 *PCB, Pass Control Back

10.21.1 Function and Requirements

The Pass Control Back command is used by a **controller** to tell a **device,** being a potential **controller,** to which address the control is to be passed back when the **device** (acting as a **controller**) sends the IEEE 488.1 interface message, take control (TCT).

Any **device** that implements this command shall have **controller** capability. Any **device** that has **controller** capability shall implement this command.

10.21.2 Syntax

The syntax for the Pass Control Back command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element optionally followed by a <PROGRAM DATA SEPARATOR> and another <DECIMAL NUMERIC PROGRAM DATA> element. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as “*PCB.” The first <DECIMAL NUMERIC PROGRAM DATA> shall round to an integer value in the range of 0 through 30. The second <DECIMAL NUMERIC PROGRAM DATA>, if present, shall round to an integer value in the range of 0 through 30.

10.21.3 Semantics

The first <DECIMAL NUMERIC PROGRAM DATA> element is rounded to an integer whose value is interpreted as the primary address of the **controller** sending the command.

The second <DECIMAL NUMERIC PROGRAM DATA> element is rounded to an integer whose value is interpreted as the secondary address of the **controller** sending the command.

A missing second address shall indicate the **controller** sending this command does not have extended addressing.

10.21.4 Related Common Commands

None.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.21.5 Standard Compliance

Mandatory for **devices** that have any **controller** capability (non-CO **devices**). See 15.1.4.

10.21.6 Error Handling

10.21.6.1 Unable to Pass Control

A **device** with no **controller** capability (CO) shall issue a Command Error if it receives the *PCB common command.

10.21.6.2 Parameters Out-of-Range

The **device** receiving the *PCB command shall issue an Execution Error if the value of either the first or second parameter is out of its specified range.

10.22 *PMC, Purge Macros Command

10.22.1 Function and Requirements

The Purge Macros command causes the **device** to delete all macros that may have been previously defined using the *DMC command. All stored macro command sequences and labels shall be removed from the **device's** memory by this command.

10.22.2 Syntax

The syntax for the Purge Macros command is defined as only a <COMMAND PROGRAM HEADER>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*PMC."

10.22.3 Semantics

Not applicable.

10.22.4 Related Common Commads

*EMC — Implementation requires this optional command.
*EMC? — Implementation requires this optional query.
*GMC? — Implementation requires this optional query.
*DMC — Implementation requires this optional command.
*LMC — Implementation requires this optional command.
*RMC — Implementation optional.

10.22.5 Standard Compliance

Optional.

10.23 *PRE, Parallel Poll Enable Register Command

10.23.1 Function and Requirements

The Parallel Poll Enable Register command sets the Parallel Poll Enable Register bits as defined in 11.6.

--`,,`,,,-`-`,,`,,`,`,,`---

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.23.2 Syntax

The syntax for the Parallel Poll Enable Register command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*PRE." The <DECIMAL NUMERIC PROGRAM DATA> value shall round to an integer in the range of 0 through 65535.

10.23.3 Semantics

The <DECIMAL NUMERIC PROGRAM DATA>, when rounded to an integer value and expressed in base 2 (binary), represents the bit values of the Parallel Poll Enable Register. See 11.6.1.3.

10.23.4 Related Common Commands

*PRE? — This is the companion query.

10.23.5 Standard Compliance

Mandatory for **devices** implementing the PP1 subset.

10.23.6 Error Handling

If the integer is outside the range of 0 to 65535 inclusive, an Execution Error shall be reported, see 11.5.1.1.5.

10.24 *PRE?, Parallel Poll Enable Register Query

10.24.1 Function and Requirements

The Parallel Poll Enable Register query allows the programmer to determine the current contents of the Parallel Poll Enable Register.

10.24.2 Query Structure

10.24.2.1 Query Syntax

The syntax for the Parallel Poll Enable Register query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*PRE?."

10.24.2.2 Response Syntax

The response syntax for the Parallel Poll Enable Register query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> value is an integer in the range of 0 through 65535.

10.24.3 Response Semantics

The <NR1 NUMERIC RESPONSE DATA> integer value expressed in base 2 (binary) represents the bit values of the Parallel Poll Enable Register. See 11.6.1.2.

10.24.4 Related Common Commands

*PRE — This is the companion command.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.24.5 Standard Compliance

Mandatory for **devices** implementing the PP1 subset.

10.25 *PSC, Power-On Status Clear Command

10.25.1 Function and Requirements

The Power-On Status Clear command controls the automatic power-on clearing of the Service Request Enable Register, the Standard Event Status Enable Register, the Parallel Poll Enable Register, and device-specific event enable registers. This command may also affect the clearing of other status registers. See 5.12. No other **device** functions shall be tied to this command. For additional IEEE 488.2 power-on requirements, see 5.12.

10.25.2 Syntax

The syntax for the Power-On Status Clear command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The <common command header> in the <COMMAND PROGRAM HEADER> is defined as "*PSC." The <DECIMAL NUMERIC PROGRAM DATA> value shall be in the range of –32767 through +32767.

10.25.3 Semantics

The value of the <DECIMAL NUMERIC PROGRAM DATA> determines whether the power-on-status-clear flag of the **device** is TRUE or FALSE.

A <DECIMAL NUMERIC PROGRAM DATA> that rounds to an integer value of zero sets the power-on-status-clear flag FALSE. Sending *PSC 0, therefore, allows instruments to assert SRQ after power-on.

A <DECIMAL NUMERIC PROGRAM DATA> that rounds to an integer value not equal to zero sets the power-on-status-clear flag TRUE. Sending *PSC 1, therefore, enables the power-on clear and disallows any SRQ assertion after power-on.

10.25.4 Related Common Commands

*PSC? — Implementation requires this optional command. This is the companion query.

10.25.5 Standard Compliance

Optional.

10.25.6 Error Handling

If the integer is outside the range of –32767 to +32767 inclusive, an Execution Error shall be reported, see 11.5.1.1.5.

10.25.7 Example

The command sequence *PSC 0;*SRE 32;*ESE 128; allows a **device** to assert SRQ upon completion of power-on.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.26 *PSC?, Power-On Status Clear Query

10.26.1 Function and Requirements

The Power-On Status Clear query allows the programmer to query the **device's** power-on-status-clear flag. A returned value of zero indicates that the Standard Event Status Enable Register, Service Request Enable Register, and the Parallel Poll Enable Register will retain their status when power is restored to the **device**. A returned value of one indicates that the registers listed above will be cleared when power is restored to the **device**, see 11.1.3.

10.26.2 Query Structure

10.26.2.1 Query Syntax

The syntax for the Power-On Status Clear query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*PSC?."

10.26.2.2 Response Syntax

The response syntax for the Power-On Status Clear query is defined as only a <NR1 DECIMAL NUMERIC RESPONSE DATA> element. The <NR1 DECIMAL NUMERIC RESPONSE DATA> shall be either a single ASCII-encoded byte for "0" (30, 48 decimal) or "1" (31, 49 decimal).

10.26.3 Response Semantics

A value of zero indicates the **device's** power-on-status-clear flag is FALSE.

A value of one indicates the **device's** power-on-status clear flag is TRUE.

10.26.4 Related Common Commands

*PSC — Implementation requires this optional command. This is the companion command.

10.26.5 Standard Compliance

Optional.

10.27 *PUD, Protected User Data Command

10.27.1 Function and Requirements

The Protected User Data command stores data unique to the **device** such as calibration date, usage time, environmental conditions, and inventory control numbers. A minimum of 63 bytes shall be provided. The size of this area shall be specified in the **device** documentation, see 4.9.

The data shall be protected by some means (such as a password or a recessed switch whose access hole can be covered by a protective sticker). The exact protection mechanism is a **device** designer decision.

Data can be stored only when the protection mechanism is disabled. An enabled protection mechanism shall cause an Execution Error to be generated when a *PUD command is received.

The contents of this memory is for data storage only and shall have no effect on **device** operation except in the response to the *PUD? query, see 10.28.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.27.2 Syntax

There are two syntax formats for the Protected User Data command:

1) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by an <ARBITRARY BLOCK PROGRAM DATA> element
2) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <STRING PROGRAM DATA> element

Devices that implement *PUD shall accept format (1) above, and may additionally accept format (2).

The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*PUD."

10.27.3 Semantics

The semantic meaning of the <ARBITRARY BLOCK PROGRAM DATA> or <STRING PROGRAM DATA> element is beyond the scope of this standard.

NOTE — The alternate <STRING PROGRAM DATA> format may only contain 7 bit ASCII data bytes, thus restricting its usage.

10.27.4 Related Common Commands

*PUD? — Implementation requires this optional command. This is the companion query.

10.27.5 Standard Compliance

Optional.

10.27.6 Error Handling

When the number of 8 bit data bytes in the <ARBITRARY BLOCK PROGRAM DATA> format or the number of 7 bit ASCII data bytes in the <STRING PROGRAM DATA> format is more than the size provided, then an Execution Error shall be generated.

An enabled protection mechanism shall cause an Execution Error to be generated when a *PUD command is received.

10.28 *PUD?, Protected User Data Query

10.28.1 Function and Requirements

The Protected User Data query allows the programmer to retrieve the contents of the *PUD storage area. See *PUD command, 10.27.

10.28.2 Query Structure

10.28.2.1 Query Syntax

The syntax for the Protected User Data query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*PUD?."

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.28.2.2 Response Syntax

The response syntax for the Protected User Data query is defined as a <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element.

10.28.3 Response Semantics

The data content of the <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> shall be the data last sent to, and accepted by, the **device** as a parameter to the *PUD command. See 10.27.

10.28.4 Related Common Commands

*PUD — Implementation requires this optional command. This is the companion command.

10.28.5 Standard Compliance

Optional.

10.29 *RCL, Recall Command

10.29.1 Function and Requirements

The *RCL command restores the current settings of a **device** from a copy stored in local memory. The scope of the *RCL command is the same as *RST and the *LRN? response. See 10.32.1 and 10.17.3. **Device** documentation shall explicitely mention the **device** settings that are restored by *RCL.

To avoid inadvertant recall of a stored setting, the use of this command may be protected by a password or other mechanism. Design and use of this mechanism is a **device** designer decision. It is recommended that any protection mechanism be clearly documented.

10.29.2 Syntax

The syntax for the Recall command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by <DECIMAL NUMERIC PROGRAM DATA> element. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*RCL." The <DECIMAL NUMERIC PROGRAM DATA> value shall be in the range of 0 through a device-specified upper bound.

10.29.3 Semantics

The <DECIMAL NUMERIC PROGRAM DATA> element shall be rounded to an integer before interpretation by the **device**.

The **device** shall recall the settings of the **device** from a memory register associated with the received integer.

The upper bound for the register number shall be the same for both the *SAV and *RCL commands, see 10.33.4.

The value zero is recommended for any special purpose register, such as a predefined power-on setup.

NOTE — If *n* is a valid register number, receiving *RCL *n* is equivalent to receiving *RST,*RCL *n*.

10.29.4 Related Common Commands

*SAV — Implementation requires this optional command.

*SDS — Implementation optional.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.29.5 Standard Compliance

Optional.

10.29.6 Error Handling

An interpreted value outside of the allowable range shall cause an Execution Error.

The **device** may generate an Execution Error when the recall of a previously initialized register is attempted, for example, after a power-on or after a receipt of a Save Default Device Settings command, *SDS. See 10.41.

An enabled protection mechanism shall cause an Execution Error to be generated when a *RCL command is received.

10.30 *RDT, Resource Description Transfer Command

10.30.1 Function and Requirements

The Resource Description Transfer command allows a Resource Description to be stored in a **device**.

This command retrieves information describing such characteristics as **device** performance, programming codes, etc., in a standard structure. This structure for Resource Descriptors is under development by the IEEE.

The data shall be protected by some means (such as a password or a recessed switch whose access hole can be covered by a protective sticker). The exact protection mechanism is a **device** designer decision.

Data can be stored only when the protection mechanism is disabled.

This memory is for data storage only and its contents shall have no effect on **device** operation except in the response to the *RDT? query, see 10.31.

10.30.2 Syntax

There are two syntax formats for the Resource Description Transfer Command:

1) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by an <ARBITRARY BLOCK PROGRAM DATA> element
2) A <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <STRING PROGRAM DATA> element

Devices that implement *RDT shall accept format (1) above, and may additionally accept format (2).

The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*RDT."

10.30.3 Semantics

The semantic meaning of the <ABRITRARY BLOCK PROGRAM DATA> or <STRING PROGRAM DATA> element is beyond the scope of this standard.

NOTE — The alternate <STRING PROGRAM DATA> format may only contain 7 bit ASCII data bytes, thus restricting its usage.

10.30.4 Related Common Commands

*RDT? — Implementation requires this command. This is the companion query.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.30.5 Standard Compliance

Optional.

10.30.6 Error Handling

10.30.6.1 Protected Data

An enabled protection mechanism shall cause an Execution Error to be generated when a *RDT command is received.

10.30.6.2 Oversized Block

A block length or string length greater than the **device** can accept shall generate an Execution Error. The **device's** documentation must specify the maximum block length.

10.31 *RDT?, Resource Description Transfer Query

10.31.1 Function and Requirements

The Resource Description Transfer query allows a Resource Description to be retrieved from a **device**. The Resource Description may be memory or in a read-write memory settable by the *RDT command.

10.31.2 Query Structure

10.31.2.1 Query Syntax

The syntax for the Resource Description Transfer query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*RDT?."

10.31.2.2 Response Syntax

The response syntax for the Resource Description Transfer query is defined as a <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA> element.

10.31.3 Response Semantics

A nonzero length block shall contain Resource Description data that is either the defined built-in Resource Description or exactly what was sent to the **device** with the last *RDT common command.

A zero-length block indicates that the **device** contains no Resource Description data.

10.31.4 Related Common Commands

*RDT — This is the companion command.

10.31.5 Standard Compliance

Optional.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.32 *RST, Reset Command

10.32.1 Function and Requirements

The Reset command performs a **device** reset. The Reset command is the third level of reset in a three-level reset strategy, see 17.1.2 and Appendix D. The Reset command shall do the following:

1) Except as explicitly excluded below, set the device-specific functions to a known state that is independent of the past-use history of the **device**. Device-specific commands may be provided to program a different reset state than the original factory-supplied one.
2) Set the macro defined by *DDT to a device-defined state, see 10.4.
3) Disable macros, see 10.8.
4) Force the **device** into the OCIS state, see 12.5.2.
5) Force the **device** into the OQIS state, see 12.5.3.

The reset command explicitly shall NOT affect the following:

1) The state of the IEEE 488.1 interface.
2) The selected IEEE 488.1 address of the **device**.
3) The Output Queue.
4) Any Event Enable Register setting, including the Standard Event Status Enable Register settings, see 11.4.2.3.4 and 11.5.1.3.4.
5) Any Event Register setting, including the Standard Event Status Register settings, see 11.4.2.2.4 and 11.5.1.2.4.
6) The power-on-status-clear flag setting.
7) Macros defined with the Define Macro Contents command.
8) Calibration data that affects **device** specifications.
9) The Protected User Data query response.
10) The Resource Description Transfer query response.
11) The Service Request Enable Register setting, see 11.3.2.4.
12) The Parallel Poll Enable Register setting, see 11.6.1.4.
13) The memory register(s) associated with *SAV.

The scope of the *LRN? response and *RCL (if implemented) is the same as *RST. See 10.17.3 and 10.29.3.

10.32.2 Syntax

The syntax for the Reset command is defined as only a <COMMAND PROGRAM HEADER>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as “*RST.”

10.32.3 Semantics

Not applicable.

10.32.4 Related Common Commands

See 10.32.1.

10.32.5 Standard Compliance

Mandatory.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.33 *SAV, Save Command

10.33.1 Function and Requirements

The *SAV command stores the current settings of the **device** in local memory. The scope of the *SAV command is the same as *RST and the *LRN? response. See 10.32.1 and 10.17.3. **Device** documentation shall explicitly mention the **device** settings that are restored by *RCL.

10.33.2 Syntax

The syntax for the Save command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*SAV." The <DECIMAL NUMERIC PROGRAM DATA> value shall be in the range of 0 through a device-specified upper bound.

10.33.3 Semantics

The <DECIMAL NUMERIC PROGRAM DATA> element shall be rounded to an integer before interpretation by the **device**.

The **device** shall save the current settings of the **device** in a memory register associated with the interpreted integer.

The upper bound for the register number shall be the same for both the *SAV and *RCL commands, see 10.29.

The value zero is recommended for any special purpose register, such as a predefined power-on setup.

10.33.4 Related Common Commands

*RCL — Implementation requires this optional command.

*SDS — Implementation optional.

10.33.5 Standard Compliance

Optional.

10.33.6 Error Handling

A value outside the allowable range shall cause an Execution Error. A **device** may have the ability to protect the information in the memory registers. If this mechanism is enabled, attempting to save the current state of the **device** shall cause an Execution Error.

10.34 *SRE, Service Request Enable Command

10.34.1 Function and Requirements

The Service Request Enable command sets the Service Request Enable Register bits as defined in 11.3.2.

10.34.2 Syntax

The syntax for the Service Request Enable command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The <common command header> in the <COMMAND PROGRAM HEADER> is defined as "*SRE." The <DECIMAL NUMERIC PROGRAM DATA> value is in the range of 0 through 255.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.34.3 Semantics

The <DECIMAL NUMERIC PROGRAM DATA>, when rounded to an integer value and expressed in base 2 (binary), represents the bit values of the Service Request Enable Register. See 11.3.2.3.

For all bits except bit 6, a bit value of one shall indicate an enabled condition. A bit value of zero shall indicate a disabled condition. See 11.3.2.3. The bit value of bit 6 shall be ignored.

10.34.4 Related Common Commands

*PSC — Determines whether the Service Request Enable Register is cleared at power-on.

*SRE? — This is the companion query.

10.34.5 Standard Compliance

Mandatory.

10.34.6 Error Handling

The **device** shall generate an Execution Error if an out-of-range parameter is received.

10.35 *SRE?, Service Request Enable Query

10.35.1 Function and Requirements

The Service Request Enable query allows the programmer to determine the current contents of the Service Request Enable Register, see 11.3.2.

10.35.2 Query Structure

10.35.2.1 Query Syntax

The syntax for the Service Request Enable query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*SRE?."

10.35.2.2 Response Syntax

The response syntax for the Service Request Enable query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> shall be in the range of 0 through 63 or 128 through 191.

10.35.3 Response Semantics

When converted to binary (base 2), the <NR1 NUMERIC RESPONSE DATA> represents the current bit values of the Service Request Enable Register. Bit 6 of the binary representation shall always be sent with value zero.

10.35.4 Related Common Commands

*SRE — This is the companion command.

10.35.5 Standard Compliance

Mandatory.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.36 *STB?, Read Status Byte Query

10.36.1 Function and Requirements

The Read Status Byte query allows the programmer to read the status byte and Master Summary Status bit.

10.36.2 Query Structure

10.36.2.1 Query Syntax

The syntax for the Read Status Byte query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*STB?."

10.36.2.2 Response Syntax

The response syntax for the Read Status Byte query is defined as a <NR1 NUMERIC RESPONSE DATA> element. The <NR1 NUMERIC RESPONSE DATA> value shall be in the range of 0 through 255.

10.36.3 Response Semantics

The <NR1 NUMERIC RESPONSE DATA> integer value expressed in base 2 (binary) represents the bit values of the Status Byte Register, see 11.2.2.2.

The MSS (Master Summary Status) bit, not the RQS message, is reported on bit-6, see 11.2.2.3.

10.36.4 Related Common Commands

None.

10.36.5 Standard Compliance

Mandatory.

10.37 *TRG, Trigger Command

10.37.1 Function and Requirements

The Trigger command is the device-specific analog of the IEEE 488.1 defined Group Execute Trigger (GET) interface message, and has exactly the same effect as a GET when received, parsed, and executed by the **device**. GET operation is discussed in detail in 6.1.4.2.5.

NOTE — that the *TRG command without *DDT implemented, see 10.4, generates a response by predefined action.

10.37.2 Syntax

The syntax for the Trigger command is defined as only a <COMMAND PROGRAM HEADER>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*TRG."

10.37.3 Semantics

Not applicable.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,`,,,,-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.37.4 Related Common Commands

*DDT — This optional command may be used to define the action of *TRG.

10.37.5 Standard Compliance

Mandatory for **devices** implementing the DT1 subset.

10.38 *TST?, Self-Test Query

10.38.1 Function and Requirements

The self-test query causes an internal self-test and places a response into the Output Queue indicating whether or not the **device** completed the self-test without any detected errors. Optionally, information on why the self-test was not completed may be contained in the response. The scope of the internal self-test shall appear in the **device** documentation, see 4.9.

The *TST? query shall not require any local operator interaction. It shall not create bus conditions that are violations to the IEEE Std 488.1-1987 [4] or IEEE Std 488.2-1992 standards. Otherwise, the scope of the self-test is completely at the discretion of the **device** designer.

Upon successful completion of *TST?, the **device** settings shall be restored to their values prior to the *TST?; set to fixed, known values that are stated in the **device** documentation; or set to values defined by the user and stored in local memory.

10.38.2 Query Structure

10.38.2.1 Query Syntax

The syntax for the Self-Test query is defined as only a <QUERY PROGRAM HEADER>. The <common query program header> in the <QUERY PROGRAM HEADER> is defined as "*TST?."

10.38.2.2 Response Syntax

The response syntax for the Self-Test query is defined as a <NR1 NUMERIC RESPONSE DATA> The <NR1 NUMERIC RESPONSE DATA> value shall be in the range of –32767 through +32767.

10.38.3 Response Semantics

A <NR1 NUMERIC RESPONSE DATA> with the value of zero indicates the self-test has completed without errors detected.

A <NR1 NUMERIC RESPONSE DATA> with the value not equal to zero indicates the self-test was not completed or was completed with errors detected.

The semantics of the nonzero response otherwise shall be completely at the discretion of the **device** designer.

10.38.4 Related Common Commands

Not applicable.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.38.5 Standard Compliance

Mandatory.

10.39 *WAI, Wait-to-Continue Command

10.39.1 Function and Requirements

The Wait-to-Continue command shall prevent the **device** from executing any further commands or queries until the no-operation-pending flag is TRUE. See 12.5.1.

NOTE — In a **device** that implements only sequential commands, the no-operation-pending flag is always TRUE.

10.39.2 Syntax

The syntax for the Wait-to-Continue command is defined as only a <COMMAND PROGRAM HEADER>. The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*WAI."

10.39.3 Semantics

Not applicable.

10.39.4 Related Common Commands

*OPC, *OPC?.

10.39.5 Standard Compliance

Mandatory.

10.40 *RMC, Remove Individual Macro Command

10.40.1 Function and Requirements

The Remove Individual Macro command removes a single macro definition from the **device**.

10.40.2 Syntax

The syntax for the Remove Individual Macro command is defined as a <COMMAND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by <STRING PROGRAM DATA>.

The <common command program header> in the <COMMAND PROGRAM HEADER> is defined as "*RMC."

10.40.3 Semantics

The <STRING PROGRAM DATA> element contains a currently defined macro label.

Any <white space> or leading colon (if compound headers are implemented), or both, before the first character of the macro label shall not be considered a part of the label. See 10.7.3

After the command is executed, receiving the label as a program header shall cause a Command Error unless the label happened to also be a device-specific command.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.40.4 Related Common Commands

*PMC,*GMC?, *LMC?, *EMC, *EMC?, and *DMC.

10.40.5 Standard Compliance

This command is optional but can only be implemented if the other macro commands were implemented.

10.40.6 Error Handling

If the macro label in the <STRING PROGRAM DATA> does not match a currently defined macro, an Execution Error shall be reported.

10.40.7 Examples

Assume *DMC “HOME”, “MOVE 0,0” defines a macro called “HOME.” Thus, if “HOME” is received as a <COMMAND PROGRAM HEADER> then “MOVE 0,0” is executed.

Receiving *RMC “HOME” causes the macro to be removed from the **device**.

Receiving *RMC“ HOME” also causes the macro to be removed from the **device**.

Receiving “HOME” as a <COMMAND PROGRAM HEADER> after the macro has been removed causes a Command Error (assuming the **device** has no device-specific HOME command).

10.41 *SDS, Save Default Device Settings Command

10.41.1 Function and Requirments

The Save Default Device Settings command initializes the contents of a save/recall register. The register contents are set to a known state that is independent of the past use history of the **device**.

10.41.2 Syntax

The syntax for the Save Default Device Settings command is defined as a <COMPOUND PROGRAM HEADER> followed by a <PROGRAM HEADER SEPARATOR> followed by a <DECIMAL NUMERIC PROGRAM DATA> element.

The <common program header> in the <COMMAND PROGRAM HEADER> is defined as “*SDS.” The <DECIMAL NUMERIC PROGRAM DATA> value shall be in the range of 0 through a device-specified upper bound.

10.41.3 Semantics

The <DECIMAL NUMERIC PROGRAM DATA> element shall be rounded to an integer before interpretation by the **device**.

The **device** shall initialize the memory register associated with the received integer.

The upper bound for the register number shall be the same as the *SAV command, see 10.33.

10.41.4 Related Common Commands

*RCL — Implementation requires this optional command.

*SAV — Implementation requires this optional command.

*RST — Implementation is mandatory.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.41.5 Standard Compliance

Optional, but can only be implemented if the *SAV and *RCL commands were implemented.

10.41.6 Error Handling

A value outside the allowable range shall cause an Execution Error. A **device** may have the ability of protecting the information in the memory registers that was saved from a previous save command. If this protection mechanism has been enabled, attempting to execute the *SDS command shall cause an Execution Error.

11. Device Status Reporting

This section specifies **device** requirements involving IEEE 488.1 Service Request and Parallel Poll functions. These requirements build upon and extend the IEEE 488.1 specifications to provide a detailed status reporting structure.

A complete model is defined for all status reporting. The IEEE 488.1 status byte is part of the model.

Specific and required status messages are defined. A device-specific status reporting model suitable for specific device-specific requirements is presented.

This section provides a method to transfer the IEEE 488.1 status byte to the **controller** using either the IEEE 488.1 serial poll or an IEEE 488.2 -defined common query. Additional common commands and queries are provided to obtain more detailed status information.

11.1 Overview

11.1.1 Operation

Fig 11-1 shows the block diagram of the IEEE 488.2 Status Reporting Structure.

IEEE 488.2 status reporting utilizes the IEEE 488.1 status byte with additional data structures and rules.

The Status Byte Register is composed of seven single-bit “summary-messages” (see Fig 11-1). Each summary-message summarizes an overlaying Status Data Structure.

Summary-messages always track the current status of the associated Status Data Structure. The summary-messages are cleared only by some action of the application program that clears the associated Status Data Structure.

Service Request Enabling and Generation determines if one or more of the seven summary-messages will generate the rsv message and, thus, cause the **device** to request service using its IEEE 488.1 Service Request function.

The Status Data Structures associated with summary-messages are defined in this section. They follow either

1) A set of registers to record **device** events and conditions (register-model)
2) Queue to record sequential status or other information (queue model)

Only two of the available seven Status Data Structures are completely defined by this standard:

1) The “Output Queue” is a queue model structure that is summarized by the Message Available (MAV) summary-message on bit 4 (DIO5, see Fig 11-2).
2) The “Standard Event Status” is a register-model structure that is summarized by the Event Status Bit (ESB) summary-message on bit 5 (DIO6, see Fig 11-2).

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.1.2 Summary of Related Common Commands

Table 11-1 lists the IEEE 488.2 -defined common commands related to this section. A short description and section references for further information are given for each command.

11.1.3 Related IEEE 488.1 — Defined Operations

Table 11-2 lists the IEEE 488.1 -defined operations related to this section. A short description and section references for further information are included with each operation.

11.2 Status Byte Register

11.2.1 Definition

The Status Byte Register contains the **device's** STB and RQS (or MSS) messages. IEEE 488.1 defines the method of reporting the STB and RQS message, but leaves the setting and clearing protocols and semantics for the STB message undefined. This standard further defines specific **device** STB summary-messages. A Master Summary Status (MSS) message is also provided which is output as bit 6 with the STB in response to a *STB? common query. See 11.2.2.2.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

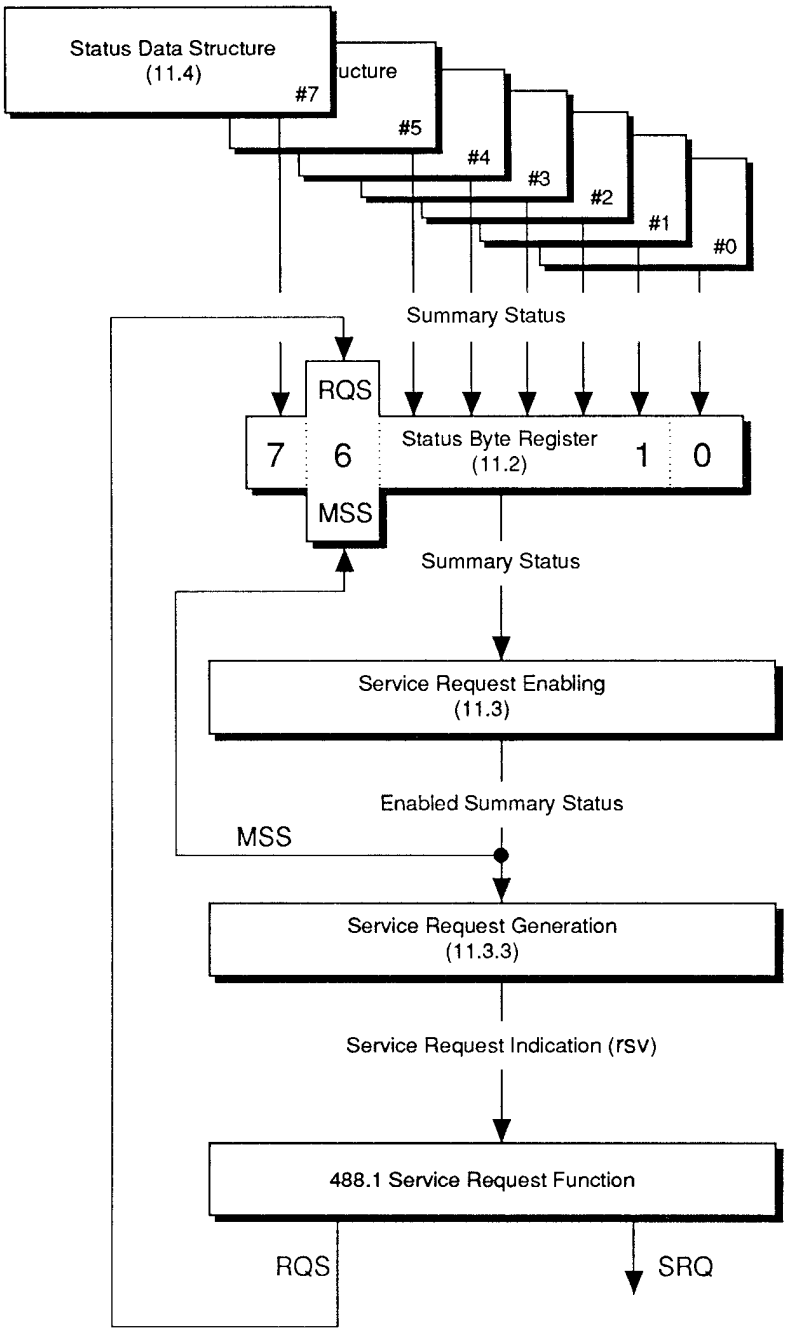


Figure 11-1—IEEE 488.2 Status Reporting Structure Overview

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

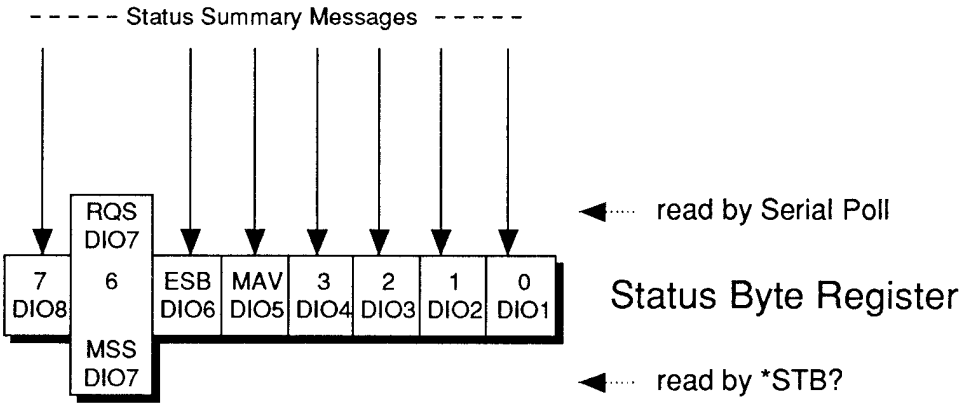


Figure 11-2—Status Byte Register

Table 11-1—Status Reporting Common Commands

*STB?	Returns an NR1, which is the value of the IEEE 488.1 status byte and the MSS (Master Summary Status) summary-message. See 10.36 and 11.2.2.2.
*OPC	Sets the "Operation Complete" event bit in the Standard Event Status Register when all selected pending **device** operations have been completed. See 10.18 and 12.5.2.
*OPC?	Places a "1" in the Output Queue when all selected pending operations are completed, which in turn causes the MAV (Message Available) summary-message to be generated. See 10.19 and 12.5.3.
*CLS	Clears all Event Registers summarized in the status byte. All Queues, except the Output Queue, that are summarized in the status byte are emptied. The **device** is forced into the Operation Complete Command Idle State and the Operation Complete Query Idle State. (See 10.3, 11.2.4, 12.5.2, and 12.5.3.2).
*ESR?	Returns an NR1, which is the value of the Standard Event Status Register. See 10.13.6 and 11.5.1.
*ESE NRf	Sets the bits of the Standard Event Status Enable Register. See 10.10 and 11.4.3.3.
*ESE?	Returns an N-R1, which is the value of the Standard Event Status Enable Register. See 10.11 and 11.4.2.3.2.
*IST?	Returns an NR1, which is the value of the IEEE 488.1 ist (individual status) local message. See 10.15 and 11.6.2.
*SRE NRf	Sets the bits of the Service Request Enable Register. See 10.34 and 11.3.2.3.
*SRE?	Returns an NR1, which is the value of the Service Request Enable Register. See 10.35 and 11.3.2.2.
*PRE NRf	Sets the bits of the Parallel Poll Enable Register. See 10.23 and 11.6.1.3.
*PRE?	Returns an NR1, which is the value of the Parallel Poll Enable Register. See 10.24 and 11.6.1.2.
*PSC NRf	Sets the power-on-status-clear flag. When set FALSE (value 0), power-on service requests are possible. See 10.25, 11.1.3, and 5.12.
*PSC?	Returns an NR1, which is the value of the state of the power-on-status-clear flag. See 10.26, 11.1.3, and 5.12.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 11-2—Status Reporting IEEE 488.1 -Defined Operations

power on	Clears the Service Request Enable Register, the Standard Event Status Enable Register, the Parallel Poll Enable Register, and device-specific event enable registers if the power-on-status-clear flag is TRUE or if the *PSC command, see 10.25.4, is not implemented. Other status registers may be affected. See 5.12.
serial poll	This is the IEEE 488.1 serial poll operation. The value of the IEEE 488.1 -defined STB and RQS messages is returned. The value of the status byte is not altered as a result of the serial poll. "rsv," however, is set FALSE by the **device** causing the RQS message to be cleared when the **device** leaves Affirmative Poll Response State (APRS).
parallel poll	This is the IEEE 488.1 parallel poll operation. A one bit status message is returned, see 11.6.1.3.

11.2.1.1 IEEE 488.2 — Defined Standard Event Status Bit (ESB) Summary-Message

The ESB summary-message is an IEEE 488.2 -defined message that appears in bit 5 of the Status Byte Register. Its state indicates whether or not one or more of the enabled IEEE 488.2 defined events have occurred since the last reading or clearing of the Standard Event Status Register, see 11.5.1.

The ESB summary-message is TRUE when an enabled event in the Standard Event Status Register is set TRUE. Conversely, the ESB summary-message is FALSE when no enabled events are TRUE. See 11.5.1.2–11.5.1.3 for detailed operation.

11.2.1.2 Standard-Defined MAV Queue Summary Message

The MAV (Message Available) summary-message is an IEEE 488.2 -defined message that appears in bit 4 of the Status Byte Register. The state of the message indicates whether or not the Output Queue (6.1.10) is empty. Whenever the **device** is ready to accept a request by the **controller** to output data bytes, the MAV summary-message shall be TRUE. The MAV summary-message shall be FALSE when the Output Queue is empty.

NOTE — In situations in which a **controller** can acquire bytes from the Output Queue faster than the **device** can supply them, the Output Queue may be emptied one or more times before the RMT is sent. The designer may allow the MAV message to remain TRUE until RMT is sent. Alternatively, the MAV message may follow the **oq-empty** message where **system** performance improvement is possible. See 6.1.10.2. It is recommended that the MAV behavior be documented.

This message is used to synchronize information exchange with the **controller**. The **controller** can, for example, send a query command to the **device** and then wait for MAV to become TRUE. The IEEE 488.1 bus is available for other use while an application program is waiting for a **device** to respond. If an application program begins a read operation of the Output Queue without first checking for MAV, all **system bus** activity is held up until the **device** responds. See Section 12 and Appendix B for **device** and application program synchronization discussion and examples.

11.2.1.3 Device-Defined Summary-Messages

Bits 0 through 3 and bit 7 are available for use by the **device** designer as summary-messages. Every device-defined summary-message shall have an associated Status Data Structure that follows one of the two models described in 11.4. This structure is either a register set for parallel event and condition reporting, or a Queue for sequential status and information reporting. The summary-message summarizes the current status of the associated Status Data Structure. That is, one or more TRUE and enabled event bits in the register-model or a nonempty Queue in the queue model is indicated by a TRUE summary-message. See 11.4.2 and 11.4.3.

11.2.2 Reading the Status Byte Register

The Status Byte Register can be read with either a serial poll or the *STB? common query. Both of these methods read the IEEE 488.1 STB message. The value sent; for the bit 6 position is, however, dependent upon the method used.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.2.2.1 Reading with a Serial Poll

When serial polled according to IEEE 488.1, the **device** shall return the 7 bit status byte plus the single bit IEEE 488.1 RQS message. The RQS message is added by the I/O Control, see 6.1.4., in accordance with the IEEE 488.1 SR state diagram. The status byte and RQS message are returned to the **controller** as a single data byte. Per IEEE 488.1, the RQS message indicates if the **device** was sending SRQ TRUE. **Devices** shall not send the END message during a serial poll.

For the purpose of this standard, the IEEE 488.1 DIO signal lines DIO1-6 and 8 correspond to bits 0-5 and 7 of the status byte register, respectively. The RQS message is sent on line DIO7 (bit 6) according to the requirements of IEEE 488.1.

A TRUE message means that the associated DIO signal line shall be asserted (pulled to a low voltage) when the status byte is sent. This convention is standard IEEE 488.1 negative logic (see IEEE Std 488.1-1987 [4], 3.2).

The STB message portion of the Status Byte Register is read nondestructively. The value of the status byte shall not be altered by a serial poll.

The **device** shall set the rsv message FALSE when polled, see 11.3.3, so that the RQS message will be FALSE if the **device** is polled again before a new reason for service, see 11.3.3, has occurred.

11.2.2.2 Reading With the *STB? Query

The *STB? common query, see 10.36, shall cause the **device** to send the contents of the Status Byte Register and the MSS (Master Summary Status) summary message, see 11.2.2.3, as a single <NR1 NUMERIC RESPONSE DATA> element.

The response shall represent the sum of the binary-weighted values of the Status Byte Register bits 0-5 and 7 (weights 1, 2, 4, 8, 16, 32, and 128 respectively) and the MSS summary message (weight 64). Thus, the response to *STB?, when considered as a binary value, is identical to the response to a serial poll except that the MSS summary message appears in bit 6 in place of the RQS message.

The *STB? common query shall not directly alter the status byte, the MSS summary message, the RQS message, or the rsv local message.

NOTE — The MAV and RQS messages may be indirectly affected, see 11.2.1.2 and 11.3.3 respectively.

11.2.2.3 Master Summary Status

The Master Summary Status (MSS) message indicates that the **device** has at least one reason for requesting service. Although the MSS message is sent in bit position 6 of the **device's** response to the *STB? query, it is not sent in response to a serial poll and should not be considered part of the IEEE 488.1 status byte.

MSS is the inclusive OR of the bitwise combination (excluding bit 6) of the Status Byte (SB) Register and the Service Request Enable (SRE) Register, see 11.3.2. That is,

MSS is defined as

(SB Register bit 0 AND SRE Register bit 0)
OR
(SB Register bit 1 AND SRE Register bit 1)
OR
(SB Register bit 2 AND SRE Register bit 2)

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

OR
(SB Register bit 3 AND SRE Register bit 3
OR
(SB Register bit 4 AND SRE Register bit 4)
OR
(SB Register bit 5 AND SRE Register bit 5)
OR
(SB Register bit 7 AND SRE Register bit 7)

Note that the MSS definition ignores the state of bit 6 in both the Status Byte Register and the Service Request Enable Register. For purposes of computing the value of MSS, implementors may choose to treat the status byte as an 8 bit value with bit 6 always zero.

11.2.3 Writing the Status Byte Register

The Status Byte Register is altered only when the state of the overlaying Status Data Structure is altered. This is illustrated in Fig 11-1.

NOTE — Changes in the status byte may affect the state of the rsv local message, MSS summary message, and the SRQ interface message, as described in 11.3.3.

11.2.4 Clearing the Status Byte Register

The *CLS common command, see 10.3, shall cause all Status Data Structures (that is, their Event Registers and Queues) to be cleared so that the corresponding summary messages are clear. The Output Queue and its MAV summary message are an exception and are unaffected by *CLS.

The entire status byte can be cleared by sending the *CLS command to the **device** after a <PROGRAM MESSAGE TERMINATOR> and before any <QUERY MESSAGE UNIT> elements. The Output Queue will be cleared of any unread messages using this method, see 6.3.2.3. With the Output Queue empty, the MAV summary message will be FALSE. The MSS message in the *STB? response will also go FALSE. The RQS message in the Status Byte Register will be FALSE unless the **device** is in Affirmative Poll Response State (APRS).

11.3 Service Request Enabling

11.3.1 Operation

Service request enabling operation is shown in Fig 11-3.

Service request enabling allows an application programmer to select which summary messages in the Status Byte Register may cause service requests. The Service Request Enable Register, illustrated in Fig 11-3, is used to select the summary messages.

11.3.2 Service Request Enable Register

11.3.2.1 Function

The Service Request Enable Register is an 8 bit register that enables corresponding summary messages in the Status Byte Register. Thus, the application programmer can select reasons for a **device** to issue a service request by altering the contents of the Service Request Enable Register.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.3.2.2 Reading the Service Request Enable Register

The Service Request Enable Register is read with the *SRE? common query, see 10.35. The response message to this query shall be an <NUMERIC RESPONSE DATA> element that represents the sum of the binary-weighted values of the Service Request Enable Register (2 raised to the power of the bit number). The value of the unused bit 6 shall always be zero.

11.3.2.3 Writing the Service Request Enable Register

The Service Request Enable Register is written with the *SRE common command, see 10.34. The *SRE is followed by a <DECIMAL NUMERIC PROGRAM DATA> element.

The <DECIMAL NUMERIC PROGRAM DATA>, when rounded to an integer value and expressed in base 2 (binary), shall represent the bit values of the Service Request Enable Register. A bit value of one shall indicate an enabled condition. A bit value of zero shall indicate a disabled condition.

The **device** shall always ignore the value of bit 6.

11.3.2.4 Clearing the Service Request Enable Register

Sending the *SRE common command with a <DECIMAL NUMERIC PROGRAM DATA> element value of zero clears the Service Request Enable Register. A cleared register does not allow status information to generate a rsv local message and thus, no service requests are issued.

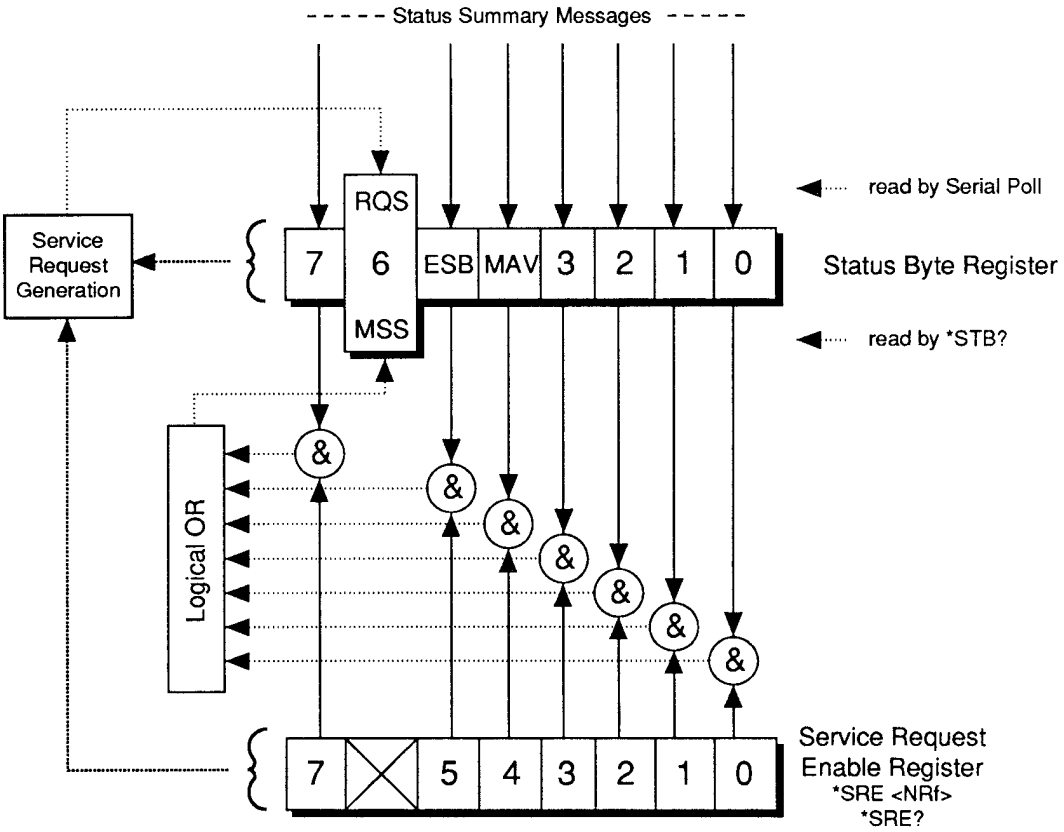


Figure 11-3—Service Request Enabling

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

If the power-on-status-clear flags is TRUE or if *PSC is not implemented, see 11.1.3, the Service Request Enable Register shall be cleared upon power-on. The *PSC command is used to set the state of the power-on-status-clear flag, see 10.25. See 5.12 for additional power-on requirements.

The Service Request Enable Register shall not be changed by the receipt of the **dcas** message, nor by any **device** condition other than as stated in this section.

11.3.3 Service Request Generation

All **devices** shall implement the Service Request function as described in IEEE Std 488.1-1987 [4]. This function provides a **device** with the capabilities of requesting service from the **controller** via the Service Request (SRQ) interface message and reporting that it has requested service via the Request Service (RQS) message, which is sent with the status byte in response to a serial poll.

The generation of service requests is controlled by the IEEE 488.1 request service (rsv) local message which is known in block diagram form in Fig. 11-4. This section describes the coupling between the status byte and the IEEE 488.1 request service (rsv) message. This section also places additional requirements (beyond those specified in IEEE Std 488.1-1987 [4] on the interaction of the IEEE 488.1 Acceptor Handshake and the rsv message. These requirements ensure that a **device** shall:

1) Assert SRQ when a previously “enabled” condition occurs.
2) Keep SRQ asserted until the **controller** has recognized the service request and polled the **device** or has taken specific action to cancel the request (for example, (*)CLS command).
3) Release SRQ when polled so that the **controller** can detect an SRQ from another **device.**
4) Assert SRQ again if another condition occurs, whether or not the **controller** has cleared the first condition.

11.3.3.1 New Reason for Service

Whenever the contents of the status byte or Service Request Enable Register are changed, the **device** must determine whether the change affects the service request state of the **device**. **Device** status transitions do not affect the state of the SRQ interface message directly. Instead, changes to the status byte and the Service Request Enable Register generate the local messages **reqt** (Request rsv TRUE) and **reqf** (Request rsv FALSE), which are inputs to the Service Request Synchronization block in Fig 11-4. The output of the Service Request Synchronization block is the local rsv message that controls the SRQ interface message as described in IEEE 488.1.

The **device** shall generate a new service request (set the **reqt** message TRUE) when

1) A bit in the status byte changes from FALSE to TRUE while the corresponding bit in the Service Request Enable Register is TRUE.
2) A bit in the Service Request Enable Register changes from FALSE to TRUE while the corresponding bit in the status byte is TRUE.
3) A bit in the status byte changes from FALSE to TRUE and the corresponding bit in the Service Request Enable Register changes from FALSE to TRUE simultaneously.

The **device** shall set the reqt message FALSE when

1) The MSS message changes from TRUE to FALSE.
2) The **device** enters the SRWS of the Set rsv State Diagram in Fig 11-4 and Tables 11-3 and 11-3.

Table 11-3—Set rsv State Diagram Mnemonics

Messages		Interface States	
reqt	Request rsv TRUE	SRIS	Service Request Idle State
reqf	Request rsv FALSE	SRWS	Service Request Wait State
rsv	request service	SRAS	Service Request Active State
		(APRS)	Affirmative Poll Response State

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 11-4—Set rsv State Diagram Message Output

Set rsv State	rsv Message Sent	Device Interaction
SRIS	F	not requesting service
SRWS	F	not requesting service
SRAS	T	requesting service

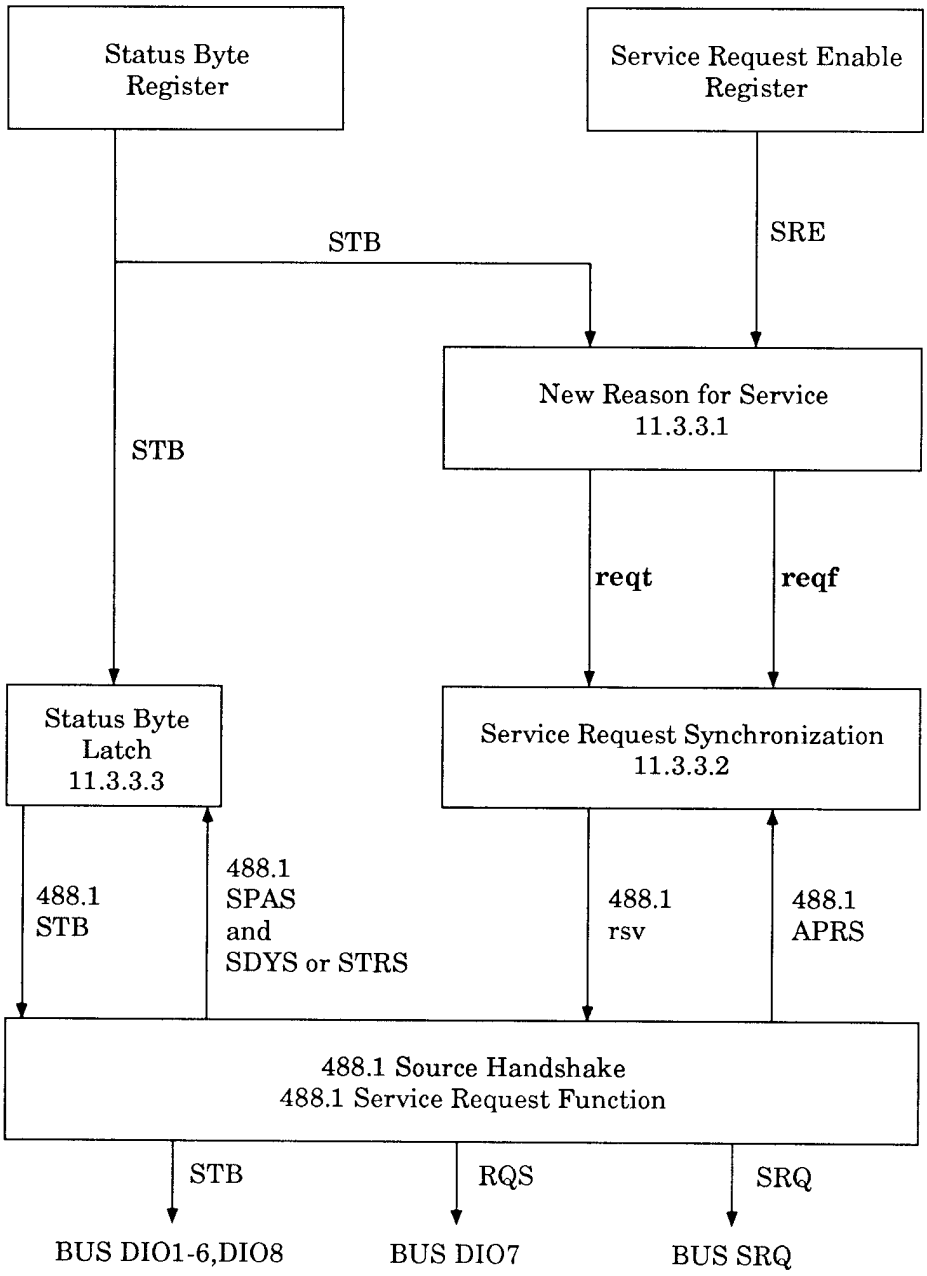


Figure 11-4—Service Request Generation

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

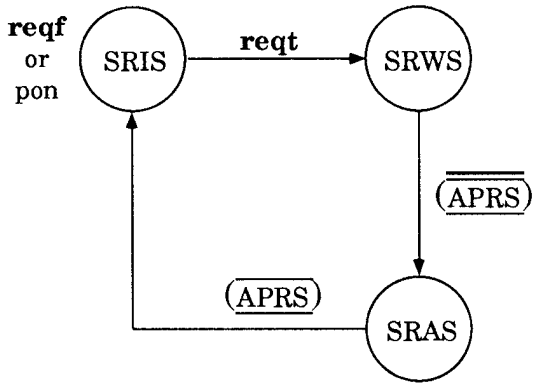


Figure 11-5—Set rsv State Diagram

The **device** shall stop requesting service (set the **reqf** message TRUE) when the MSS message changes from TRUE to FALSE.

The **device** shall set the **reqf** message FALSE when it enters Service Request Idle State (SRIS) of the Set rsv State Diagram in Fig 11-5 or at power-on.

The **device** may also set **reqt** TRUE whenever the contents of the status byte are changed and MSS is TRUE. Because extraneous service requests may be generated, this method is not recommended. It is permitted, however, to accommodate certain commercially-available IEEE 488.1 interface circuits. See 11.3.3.4 on implementation techniques.

In general, the **controller** application program must never assume that SRQ indicates that a new reason for service has occurred, but only that a new reason for service may have occurred; and that the application program should check the **device** status byte to determine whether this is indeed the case.

11.3.3.2 Service Request Synchronization

The Set rsv State Diagram, shown in Fig 11-5, synchronizes changes to the **device's** status byte and Service Request Enable Register with the state of the IEEE 488.1 SR function. It ensures that the **device** does not change the SRQ or RQS interface messages while being polled by the **controller**, but that the **device** will "remember" that the status change occurred and will change the state of SRQ and RQS on exit from Affirmative Poll Response State (APRS).

At power on, or when the **reqf** local message is TRUE, the **device** shall enter Service Request Idle State (SRIS). While in this state, the **device** shall send the rsv local message FALSE.

When the **device** sets the **reqt** local message TRUE, the **device** shall enter Service Request Wait State (SRWS). The **device** shall remain in SRWS and continue to send the rsv local message FALSE until the pon or **reqf** local messages become TRUE or the **device** leaves the IEEE 488.1 APRS. Staying in SRWS ensures that the **device** will not change the RQS or SRQ interface messages from FALSE to TRUE while it is being polled by the **controller**.

If the **device** is not in APRS, or when it leaves APRS, it shall enter Service Request Active State (SRAS). While in this state, the **device** shall send the rsv local message TRUE. The **device** shall remain in this state until the pon or **reqf** local messages become TRUE or the **device** enters APRS.

Upon entry into APRS, the **device** shall return to SRIS and set rsv FALSE, indicating that it has been polled. Note that the **device** will continue to send the RQS interface message TRUE until it has left APRS, according to IEEE Std 488.1-1987 [4].

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,`,,`-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.3.3.3 Status Byte Latch

If the contents of the Status Byte Register were to be placed directly onto the bus when the **device** was polled, a change in the status byte while the **device** is in IEEE 488.1 Serial Poll Active State (SPAS) could cause the **controller** to read the status byte incorrectly, due to the changing state of the bus data I/O lines. To prevent this problem, **devices** shall latch the contents of the **device** status byte and RQS message as presented on the DIO lines while the **device** is in IEEE 488.1 SPAS and the IEEE 488.1 source handshake is in Source Delay State (SDYS) or Source Transfer State (STRS). The latch guarantees that the status byte will not change on the bus just as the IEEE 488.1 SR diagram assures that the RQS message will not change. This requirement does not imply that the contents of the Status Byte Register may not be changed during this time, only that the value presented to the **controller** on DIO1-8 will be stable. In Fig 11-4, the input from the IEEE 488.1 Source Handshake/IEEE 488.1 Service Request block determines whether or not the Status Byte Latch is transparent.

11.3.3.4 Implementation Techniques

Many commercially available IEEE 488.1 integrated circuits implement the Set rsv State Diagram in place of, or in addition to, providing direct control over the rsv local message. These circuits also include a status byte buffer that allows the IEEE 488.1 control circuit to respond to a serial poll without the intervention of **device** firmware. This section describes how the requirements of the IEEE 488.2 status reporting model may be met using such circuits. It does not sanction the use of specific components or prohibit alternate implementations that meet the requirements of this standard, but is meant as a discussion of common implementation techniques and pitfalls.

11.3.3.4.1 Preferred Implementation

The preferred implementation, from a **device** designer's point of view, would be to do the following when the contents of the status byte buffer are updated:

1) Set **reqt** TRUE if New Reason for Service is TRUE.
2) Set **reqf** TRUE if MSS is FALSE.

Here, “New Reason for Service” is an additional argument to the command, generated by the **device** firmware in accordance with 11.3.3.1. This implementation avoids the race condition of 11.3.3.4.2 and the spurious service requests of 11.3.3.4.3, but requires that nine bits of information be specified with the command (seven bits of status, MSS, and New Reason for Service)

11.3.3.4.2 Allowed Independent Control of STB, reqt, and reqf

Some integrated circuits allow the contents of the circuit’s status byte buffer to be updated at any time without changing the state of rsv. Such circuits have three inputs:

1) A command that updates the contents of the circuit’s status byte buffer.
2) A command that sets **reqt** TRUE.
3) A command that sets **reqf** TRUE.

TRUE to FALSE transitions of **reqt** and **reqf** are generated internally by the circuit according to the Set rsv State Diagram. Inputs (2) and (3) may be implemented as separate commands or as a single command with one argument.

When the **device** status byte or Status Enable Register changes, the **device** firmware must

1) Update the circuit's status byte buffer.
2) If there is a New Reason for Service, see 11.3.3.1, set **reqt** TRUE.
3) If there is no reason for service (MSS FALSE), set **reqf** TRUE.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Note that there is a race between the **device's** firmware setting **reqt** or **reqf** TRUE and the **controller's** polling the **device** in response to a previous service request. If the **controller** polls the **device** after it has updated the status byte buffer but before it has set **reqt** TRUE, the **device** will generate a service request after the **controller** has read the new status byte. If the **controller** polls the **device** after it has updated the status byte buffer but before it has set **reqf** TRUE, the **controller** will receive the new status byte (which shows no reason for service request) with RQS TRUE (indicating a request for service). This race does not cause problems in practice, as the **controller** receives the most recent **device** status in either case. Implementations that set **reqt** TRUE before updating the status byte buffer are not allowed, as they could cause a violation of 11.3.3, rule (2).

11.3.3.4.3 Allowed Coupled Control of STB, reqt, and reqf

Some integrated circuits require that either **reqt** or **reqf** be set TRUE each time the status byte buffer is updated. Such circuits have one input. When the contents of the status byte buffer are updated, the circuit sets

1) **reqt** TRUE if MSS is TRUE, or
2) **reqf** TRUE if MSS is FALSE

This implementation is allowed by this standard (next to last paragraph of 11.3.3.1), but can generate a service request when a bit in the status byte changes, even if that bit has not been enabled to cause a service request.

11.3.3.4.4 Other Allowed Implementations

Other implementation techniques that satisfy the requirements of this standard may be used.

11.4 Status Data Structures

11.4.1 Overview

All **device** Status Data Structures shall follow either the register model or the queue model as defined in 11.4. The Status Data Structures provide a common way for the **device** designer to perform status reporting. Each status data structure has a single “output,” a summary message that summarizes the structure's state. A FALSE summary status means that there is no status to report and a TRUE summary status indicates that there is status to report.

11.4.2 Status Data Structure — Register Model

The register model Status Data Structure allows the **device** designer to summarize multiple events in a single summary message in the Status Byte Register. Fig 11-6 illustrates a generalized register model Status Data Structure.

11.4.2.1 Condition Registers

11.4.2.1.1 Function

A condition is a **device** state that is either TRUE or FALSE. A Condition Register reflects these states in its condition bits. A Condition Register may range in length from i to a maximum of 16 bits and may contain unused bits. Unused bits shall return a zero value when read.

NOTE — Some **controllers** may have difficulty reading a 16 bit unsigned integer into their internal format.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

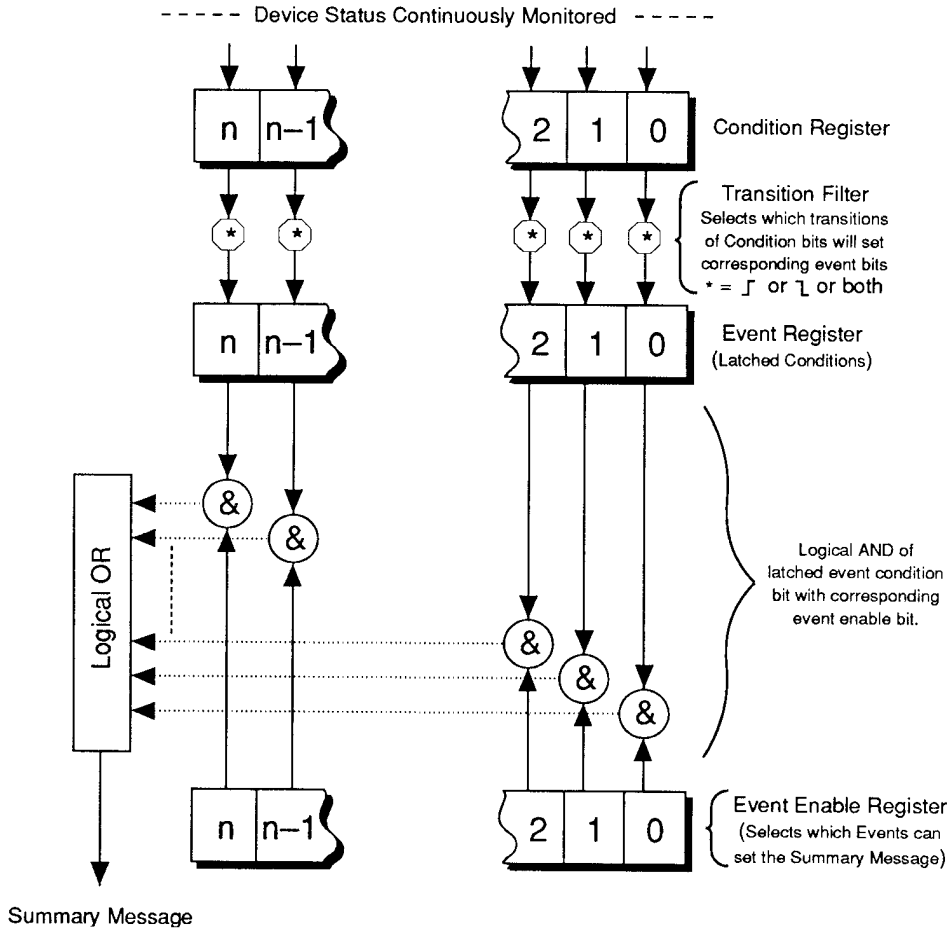


Figure 11-6—Status Data Structure — Register Model

Typically, the condition bits represent **device** states. Alternately, the condition bits may represent summary information from an overlaying secondary Status Data Structure. This secondary Status Data Structure shall be one of the Status Data Structures defined in 11.4, either the register model or the queue model. This overlaying technique may be extended to additional layers as required.

Implementation of Condition Registers is optional, since the sole purpose of a Condition Register is to group related **device** states or overlaying summary messages so that they may be read together.

This standard does not define any specific Status Data Structure that contains a Condition Register.

11.4.2.1.2 Reading Condition Registers

The **device** may provide device-specific commands to read any or all of the **device's** Condition Registers. Reading a Condition Register shall not change its contents.

11.4.2.1.3 Writing Condition Registers

No device-specific commands shall directly write to a Condition Register. Only changes in **device** state or in an overlaying secondary Status Data Structure may alter the contents of a Condition Register.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.4.2.1.4 Clearing Condition Registers

No device-specific commands shall directly clear a Condition Register. Only changes in **device** state or clearing of an overlaying secondary Status Data Structure may clear a condition bit.

11.4.2.2 Event Registers

11.4.2.2.1 Function

Event Registers capture changes in conditions. Each event bit in an event register shall correspond to a condition-bit in a Condition Register or (if there is no Condition Register) to a specific condition in the **device**. An event shall become TRUE when the associated **device** condition makes certain device-defined transitions.

The Event Register definition guarantees that the application cannot miss an event that occurs between the reading and clearing of an Event Register. An Event Register bit (event bit) shall be set TRUE when an associated event occurs. These bits, once set, are "sticky." That is, they cannot be cleared even if they do not reflect the current status of a related condition, until they are read by the application.

This standard allows the following three transition criteria for setting event bits TRUE (see transition filter in Fig 11-5):

1) *Positive transition*. The event becomes TRUE when its associated condition makes a FALSE to TRUE transition only.
2) *Negative transition*. The event becomes TRUE when its associated condition makes a TRUE to FALSE transition only.
3) *Positive or negative transition*. The event becomes TRUE when its associated condition makes either a FALSE to TRUE or a TRUE to FALSE transition.

The **device** designer shall specify which of the above transition criteria is to be used for each event bit. The **device** may provide device-specific commands to change the transition associated with each condition and event bit.

In all cases, the change of a **device** condition shall not cause an event bit to go from TRUE to FALSE.

Event Registers may range in length from 1 to a maximum of 16 bits and may contain "unused" bits. Unused bits shall be zero when the register is read.

NOTE — Some **controllers** may have difficulty reading a 16 bit unsigned integer into their internal format.

This standard completely specifies only one Event Register, the Standard Event Status Register, see 11.5.1.

11.4.2.2.2 Reading Event Registers

The **device** may provide device-specific queries to read any or all of the **device's** Event Registers. The only common query that reads an Event Register is the *ESR? query, which reads the Standard Event Status Register, see 10.12 and 11.5.1.2.2.

11.4.2.2.3 Writing Event Registers

No device-specific command shall allow the application programmer to directly write bits in an Event Register. Commands may, however, clear the entire register.

11.4.2.2.4 Clearing the Event Registers

Event Registers shall be cleared when read by a query. Only the Event Register(s) queried shall be cleared.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Additionally, specific commands, in contrast to queries, may be implemented to clear (and not read) Event Registers.

The *CLS common command is a mandatory command, see 10.3, that clears all Event Registers. All device-defined Event Registers as well as the Standard Event Status Register, see 11.5.1, shall be cleared by *CLS.

Event Registers may be cleared at power-on. Event Registers shall not be cleared in any way other than as stated previously.

11.4.2.3 Event Enable Registers

11.4.2.3.1 Function

Event Enable Registers select which event bits in the corresponding Event Register will cause a TRUE summary message when set. Each event bit shall have a corresponding enable bit in the Event Enable Register. By use of the enable bits, an application programmer can program a **device** to request service for a single event or an inclusive OR of any group of events.

Each Event Enable Register shall be the same length as the corresponding Event Register. Any unused bits in the Event Enable Register shall correspond with unused bits in the Event Register. The value of unused bits shall be zero when the Event Enable Register is read and shall be ignored when written to by commands.

This standard completely specifies only one Event Enable Register, the Standard Event Status Enable Register, see 11.5.1.3.

11.4.2.3.2 Reading Event Enable Registers

The **device** may provide device-defined queries to read any or all of the **device's** Event Enable Registers. If a **device** provides a command to write to an Event Enable Register and a query to read the same Event Enable Register, the <QUERY HEADER> shall be formed by placing a "?" after the <COMMAND HEADER> that sets the register, see 11.4.2.3.3.

The query response shall be returned as an <NR1 NUMERIC RESPONSE DATA> element. Binary-weighting of the bits shall be used to form the NR1 value. An unused bit shall have the value zero.

The only common query that reads an Event Enable Register is *ESE?, which reads the Standard Event Status Enable Register, see 10.11 and 11.5.1.3.2.

11.4.2.3.3 Writing Event Enable Registers

The **device** may provide device-specific commands to write to any or all of the **device's** Event Enable Registers. If such a command is implemented, it shall follow the format as described in this section.

The command format shall consist of a device-defined header followed by a <DECIMAL NUMERIC PROGRAM DATA> element. The **device** may also optionally accept an <EXPRESSION DATA) element that evaluates to a <DECIMAL NUMERIC PROGRAM DATA> element.

The <DECIMAL NUMERIC PROGRAM DATA>, when rounded to an integer value and expressed in base 2 (binary), shall represent the individual bit values of the Event Enable Register. A bit value of one shall indicate an enabled condition. A bit value of zero shall indicate a disabled condition.

Additional device-specific commands that use other formats are also allowed. If no command is provided to write to an Event Enable Register, then every used bit in the register shall have a value of one and every unused bit shall have a value of zero.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The only defined common command that writes to an Event Enable Register is the *ESE common command, which writes to the Standard Event Status Enable Register, see 10.10.

11.4.2.3.4 Clearing Event Enable Registers

If the **device** provides a device-specific command to write to an Event Enable Register, see 11.4.2.3.3, the register may be cleared by sending that command with an argument of zero (NRf format).

Event Enable Registers shall be cleared at power-on if the optional *PSC common command is not implemented or if power-on-status-clear flag is TRUE, see 5.12.

No other **device** or interface conditions, including receipt of the **dcas** message, shall alter any Event Enable Register.

11.4.3 Status Data Structure — Queue Model

The queue model Status Data Structure allows the **device** designer to report sequential status or other information. The presence of such information in a Queue is summarized in a summary message.

One Status Data Structure using the queue model, the Output Queue with its associated MAV summary message (bit 4 of the status byte), is defined by this standard, see 11.5.2.

11.4.3.1 Function

Fig 11-7 represents the queue model for a Status Data Structure. The Queue is a data structure containing a sequential list of information. The Queue is empty when all information has been read from the list.

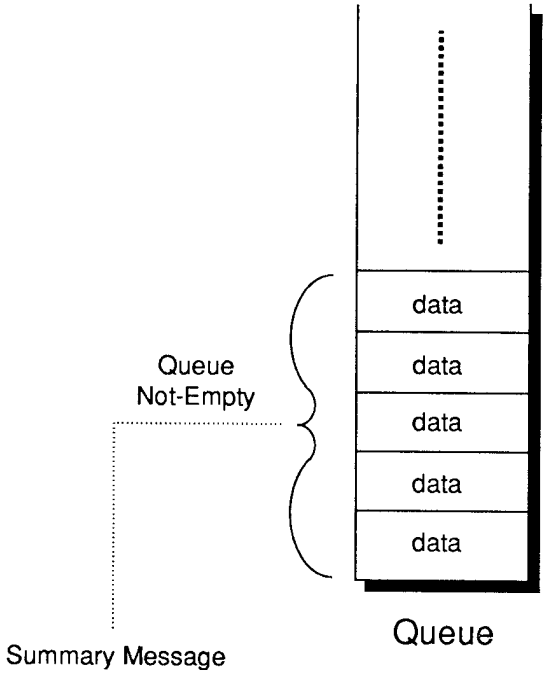


Figure 11-7—Status Data Structure — Queue Model

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Items may be placed in the Queue in any order. Items shall be removed from the Queue as they are read. The Queue may be cleared under the conditions specified in 11.4.3.4. The associated summary message is TRUE if the Queue is not empty and FALSE if it is empty.

All <RESPONSE MESSAGE UNIT> elements in a Queue (other than the Output Queue) shall be of the same type, see Fig 8-4. The specific content of the data element sent is **device** defined. This rule assures that an application program can know, in advance, the format of any data read.

For example, a Queue could contain a list of <CHARACTER RESPONSE DATA> elements. Another Queue could contain a list of <NR3 NUMERIC RESPONSE DATA> elements. Yet another Queue could contain a message unit with the two elements separated by a comma as: <NR1 NUMERIC RESPONSE DATA>,<STRING RESPONSE DATA>.

The order that the items are read from a Queue (other than the Output Queue) is beyond the scope of this standard. A device-defined Queue need not be read in a first-in, first-out (FIFO) manner.

The **device** designer may impose a maximum length on the Queue structure. Rules governing a Queue overflow condition, other than the Output Queue, are beyond the scope of this standard.

11.4.3.2 Reading a Queue

A Queue (other than the Output Queue) may be read by optional device-defined queries. Such device-defined queries shall cause the item read to be removed from the Queue and placed into the Output Queue.

11.4.3.3 Writing to a Queue

No Queue shall be written to directly by <PROGRAM MESSAGE> elements. Except for the Output Queue, the Queue represents **encoded device** information, such as a **device** status history or other information as defined by the **device** designer.

11.4.3.4 Clearing a Queue

Queues, except the Output Queue, shall be cleared when any of the following occur:

1) Receipt of the *CLS command.
2) Reading all the items in the Queues. Only the Queue that is queried shall have data removed.
3) Other device-specific means.

NOTE — The Output Queue is a special case and is not cleared by *CLS, but can be cleared under other conditions; see 6.3.2.3 and 10.3.3, including method (2)

11.5 Standard Status Data Structure

Fig 11-8 represents an overview diagram of the Standard Event Status — Status Data Structures. The Standard Event Status Register model is discussed in detail in 11.5.1. Output Queue requirements are given in 11.5.2.

11.5.1 Standard Event Status Register Model

Fig 11-9 represents the operation of the Standard Event Status Register Model. This data structure is a specific implementation of the general Status Data Structure register model, see 11.4.2, and is required by all **devices**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.1.1 Standard Event Status Register Bit Definitions

This standard assigns specific IEEE 488.2 -defined events to specific bits in the Standard Event Status Register.

11.5.1.1.1 Bits 15 through 8 — Reserved

Standard Event Status Register bits 15 through 8 are reserved for possible future use by IEEE. These bit values shall be reported as zero.

11.5.1.1.2 Bit 7 — Power On (PON)

This event bit indicates that an off-to-on transition has occurred in the **device's** power supply.

11.5.1.1.3 Bit 6 — User Request (URQ)

This event bit indicates that one of a set of local controls, see 5.6.1.3, defined by the **device** designer as a User Request control has been activated. The setting of this event-bit shall occur regardless of the IEEE 488.1 Remote/Local state of the **device**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,`,,,`-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

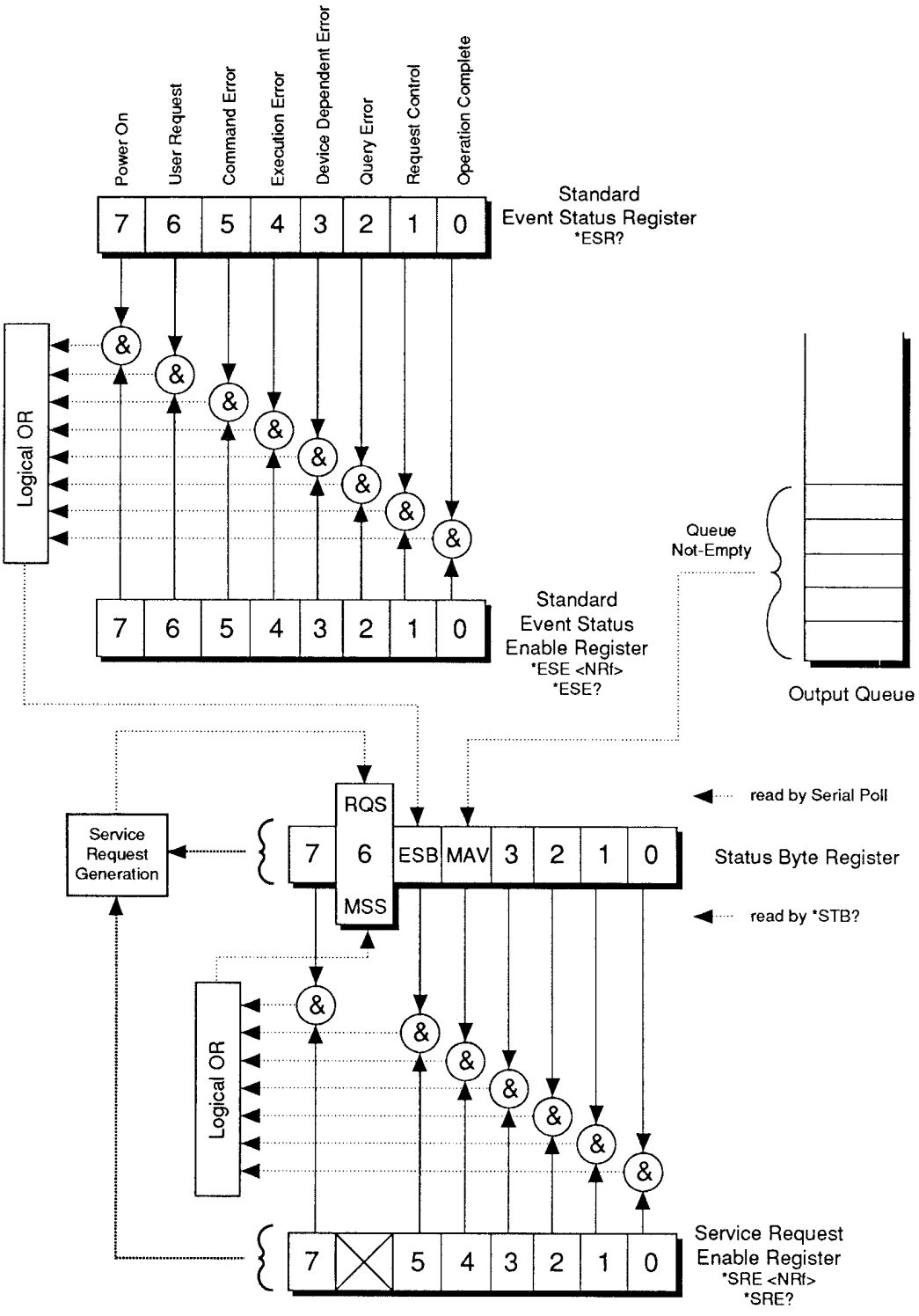


Figure 11-8—Standard Status Data Structures Overview

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

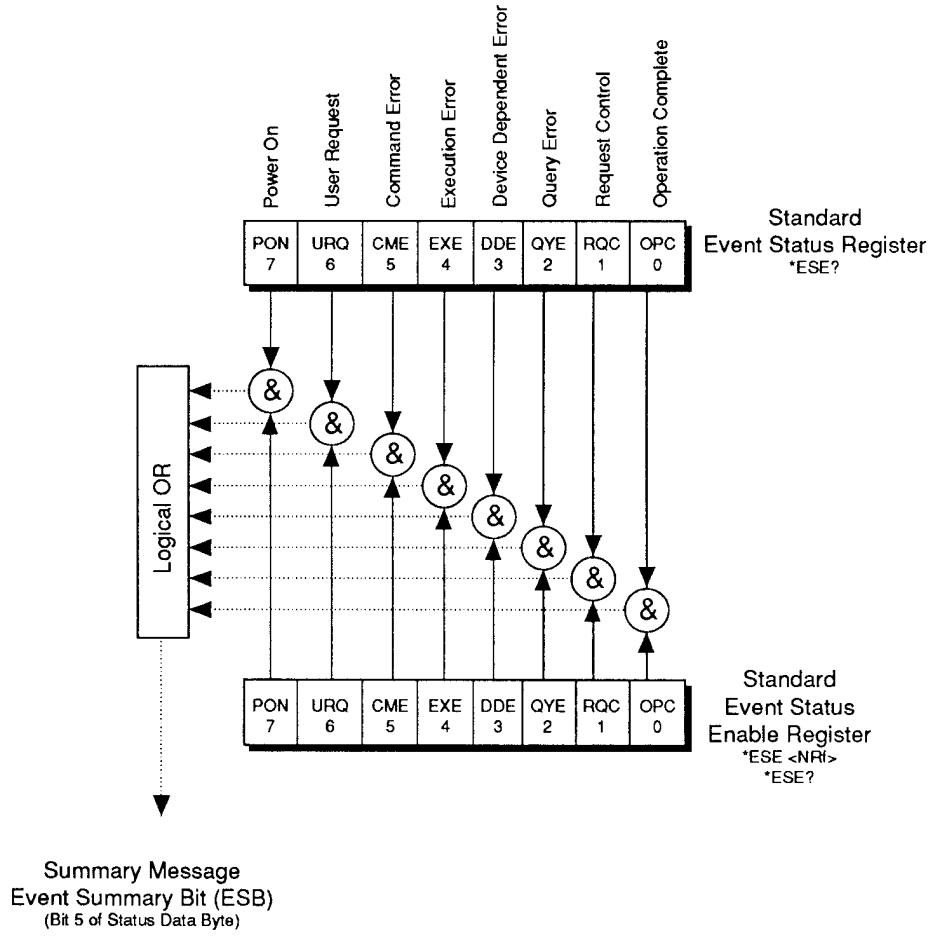


Figure 11-9—Standard Event Status Register Model

11.5.1.1.4 Bit 5 — Command ERROR (CME)

Command Errors are detected by the Parser, see 6.1.6. This event bit indicates that one of the following events has occurred:

1) An IEEE 488.2 syntax error has been detected by the parser. That is, a controller-to-device message was received that is in violation of this standard. Possible violations include a data element that violates the **device** listening formats or whose type is unacceptable to the **device**, see 7.1.2.2.
2) A semantic error has occurred indicating that an unrecognized header was received. Unrecognized headers include incorrect device-specific headers and incorrect or unimplemented IEEE 488.2 common commands, see Section 10.. A valid macro label, which is not the same as a device-specific <COMMAND PROGRAM HEADER> or <QUERY PROGRAM HEADER>, that is received by a **device** with its macros disabled shall be considered an unrecognized header.
3) A Group Execute Trigger (GET) was entered into the Input Buffer inside of a <PROGRAM MESSAGE>, see 6.1.6.1.1 and 6.4.3.

When a **device** detects a Command Error, parser synchronization may be lost. See 6.1.6.1.1 for discussion of parser action after a Command Error.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The Command Error bit shall not be set to report any other device-specific condition. Events that are reported as Command Errors shall not also be reported as Execution Errors, Query Errors, or Device-Specific Errors. See other bit definitions in this section.

11.5.1.1.5 Bit 4 — Execution ERROR (E)

Execution Errors are detected by the Execution Control Block, see 6.1.7. This event bit indicates that:

1) A <PROGRAM DATA> element following a header was evaluated by the **device** as outside of its legal input range or is otherwise inconsistent with the **device's** capabilities.
2) A valid program message could not be properly executed due to some **device** condition.

Following an Execution Error, the **device** shall continue parsing the input stream. The **device** may continue executing parsed commands or the **device** may discard parsed commands. **Devices** shall resume execution of parsed commands after a <PROGRAM MESSAGE TERMINATOR>. It is recommended that this choice be documented.

Execution Errors shall be reported by the **device** after rounding and expression evaluation operations have taken place. Rounding a numeric data element, for example, shall not be reported as an Execution Error.

Events that generate Execution Errors shall not also generate Command Errors, Query Errors, or Device-Specific Errors. See other bit definitions in this section.

NOTE — The **device** designer has the responsibility to ensure that **devices** incorporate effective checking to prevent execution of commands after an Execution Error that could result in undesireable conditions. Documentation should indicate any known conditions that cannot be checked by the **device**.

11.5.1.1.6 Bit 3 — Device-Specific ERROR (DDE)

Device-Specific Errors are detected by Device Functions, see 6.1.8. This event bit indicates that an error has occurred that is neither a Command Error, a Query Error, nor an Execution Error.

A Device-Specific Error is any executed **device** operation that did not properly complete due to some condition, such as overrange.

Following a Device-Specific Error, the **device** shall continue to process the input stream.

Device-Specific Errors are to be used at the discretion of the **device** designer.

Events that generate Device-Specific Errors shall not also generate Command Errors, Query Errors, or Execution Errors.

11.5.1.1.7 Bit 2 — Query ERROR (QYE)

Query Errors are detected by the Output Queue Control, see 6.1.10. This event bit indicates that either

1) An attempt is being made to read data from the Output Queue when no output is either present or pending, or
2) Data in the Output Queue has been lost.

See 6.5.7 for a complete description.

The Query Error bit shall not be set to report any other condition. Events that generate Query Errors shall not also generate Execution Errors, Command Errors, or Device-Specific Errors.

--`,,`,,`-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

11.5.1.1.8 Bit 1 — Request Control (RQC)

This event bit indicates to the **controller** that the **device** is requesting permission to become the active IEEE 488.1 controller-in-charge.

11.5.1.1.9 Bit 0 — Operation Complete (OPC)

This event bit is generated in response to the *OPC command. It indicates that the **device** has completed all selected pending operations. See 12.5.2 for a discussion of **device** synchronization using this event bit.

11.5.1.2 Standard Event Status Register Operation

11.5.1.2.1 Function

Standard Event Status Register operation follows the rules for event registers in the general Status Data Structure register model described in 11.4.2.

11.5.1.2.2 Reading

The Standard Event Status Register is destructively read (that is, read and cleared) with the *ESR? common query, see 10.12.

11.5.1.2.3 Writing

The Standard Event Status Register cannot be written remotely except to clear it.

11.5.1.2.4 Clearing

The Standard Event Status Register shall be cleared by

1) A*CLS command
2) A *ESR? query

A **device** undergoing a power-on sequence may initially clear the Standard Event Status Register and then record any subsequent events during the **device's** power-on sequence, including setting the PON event bit.

The Standard Event Status Register shall not be cleared by any other **device** condition.

11.5.1.3 Standard Event Status Enable Register Operation

11.5.1.3.1 Function

The Standard Event Status Enable Register allows one or more events in the Standard Event Status Register to be reflected in the ESB summary-message bit. This register follows the rules of operation of the Status Data Structure register model for Event Enable Registers, see 11.4.2.3.

This register is defined for 8 bits, each corresponding to the bits in the Standard Event Status Register. Bits 8 through 15 are reserved by the IEEE for future use.

11.5.1.3.2 Reading

The Standard Event Status Enable Register is read with the common query, *ESE?. Data is returned as a binary-weighted <NR1 NUMERIC RESPONSE DATA>, see 10.11.

11.5.1.3.3 Writing

The Standard Event Status Enable Register is written to by the common command, *ESE. Data is encoded as <DECIMAL NUMERIC PROGRAM DATA>, see 10.10.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.1.3.4 Clearing

The Standard Event Status Enable register shall be cleared by the following:

1) Sending *ESE with a data value of zero.
2) A power-on event if the power-on-status-clear flag is TRUE or the *PSC command is not implemented. See 10.25 for further details.

No other **device** or interface conditions, including receipt of the **dcas** message, shall alter this register.

The Standard Event Status Enable Register shall specifically not be affected by the following:

1) An IEEE 488.1 Device Clear function state change
2) Sending the *RST common command

11.5.2 Standard Queue Model

11.5.2.1 Function

The Output Queue stores response messages until they are read. The availability of output is summarized by the message available (MAV) summary message (bit 4 of the status byte). See Fig 11-10. See MAV implementation note, 11.2.1.2.

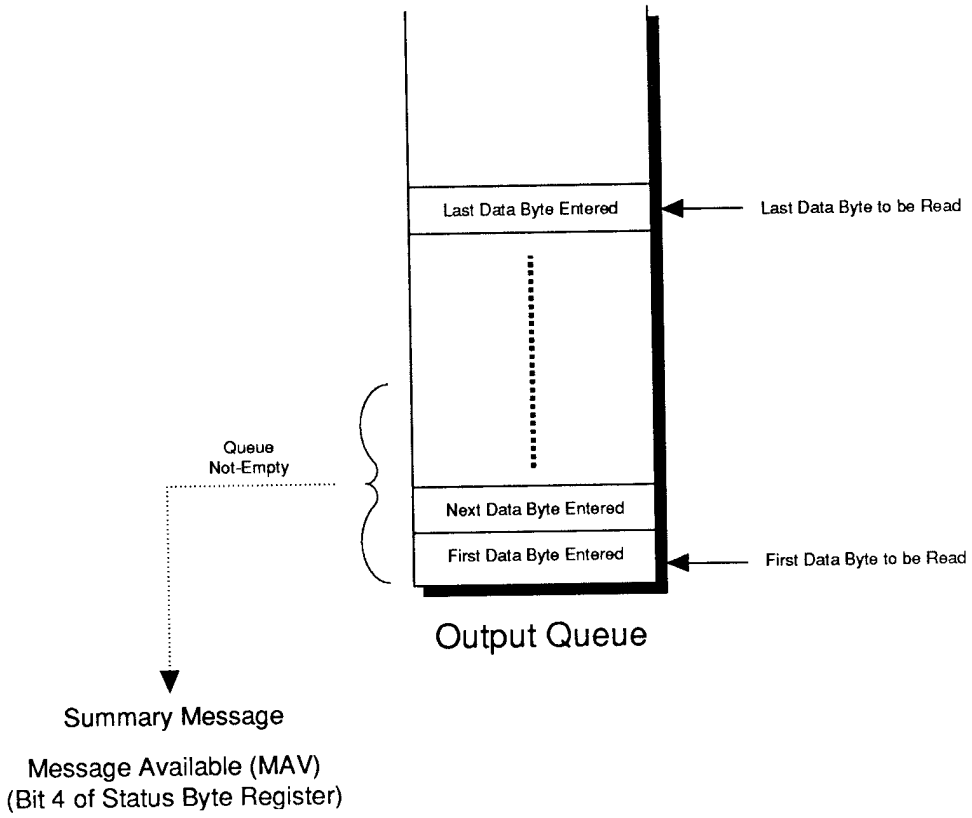


Figure 11-10—Standard Queue Model

NOTE — MAV becomes TRUE whenever data is ready and not necessarily in the Output Queue; see 6.4.5.4 and 6.1.10.2.1.

The MAV summary message is used to synchronize information exchange with the **controller**. See Section 12. for a discussion of **device** synchronization and Section 6. for details on the operation of the Output Queue.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.2.2 Reading

The Output Queue shall be a first-in, first-out (FIFO) Queue. The Queue is read by handshaking bytes out while the **device** is in Talker Active State (TACS).

11.5.2.3 Writing

The Output Queue interfaces to the **system interface** only through the protocols defined in Section 6..

11.5.2.4 Clearing

The Output Queue shall be cleared in accordance with the protocols of Section 6.. The *CLS command specifically shall NOT clear the Output Queue except as required in 6.3.2.3 for any <PROGRAM MESSAGE>.

11.6 Parallel Poll Response Handling

This section describes a method for generating and controlling the IEEE 488.1 ist (individual status) local message. This message is the status bit sent during a parallel poll operation. If a **device** has PP1 (complete) capability, the **device** shall implement the data structure and associated common commands of this section.

11.6.1 Parallel Poll Enable Register

11.6.1.1 Function

The Parallel Poll Enable Register shown in Fig 11-11 is an 8-16 bit wide register. Each bit in this register corresponds to a bit in the status byte or a device-defined condition. All the bits in the status byte must be used. Up to eight more conditions in the **device** may be associated with the upper eight bits of the Parallel Poll Enable Register.

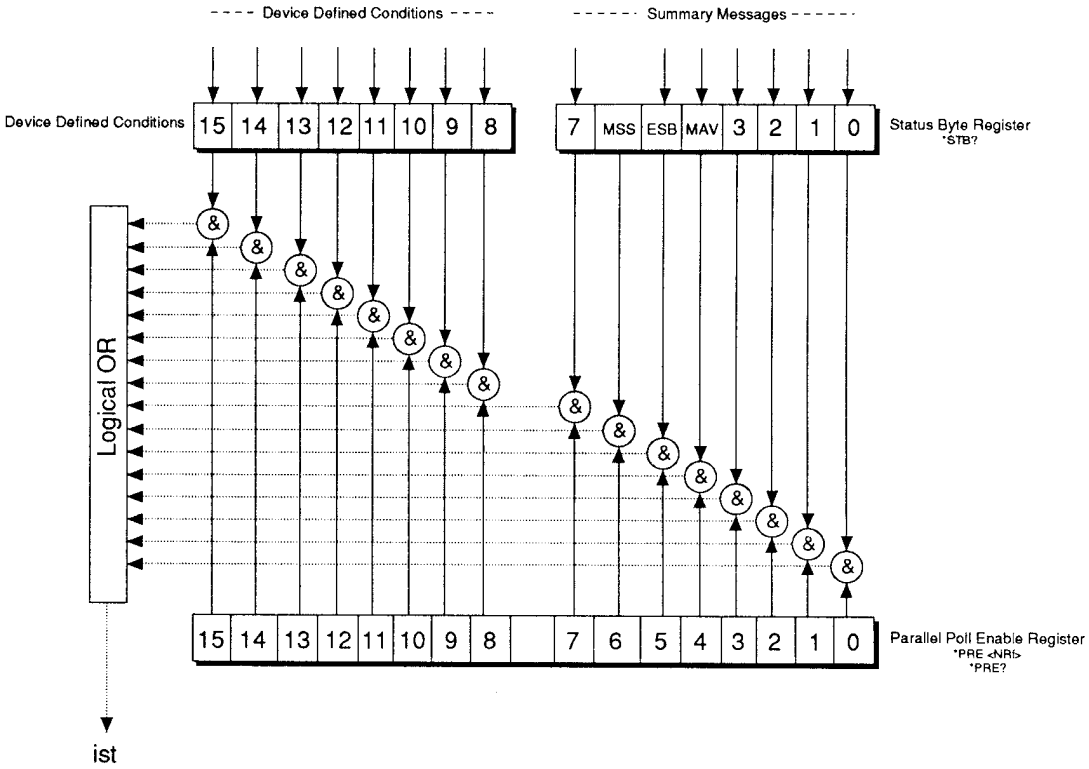


Figure 11-11—Parallel Poll Response Handling Data Structure

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Each bit in the Parallel Poll Enable Register is ANDed with its corresponding condition or summary bit. The resulting bits are ORed together to generate ist. Using the Parallel Poll Enable Register allows any single bit or combination of bits to control list.

The Master Summary Status (MSS) message is used in place of RQS to report bit 6 of the Status Byte Register. MSS is the message returned with the status byte in response to a *STB? common query, see 11.2.2.2.

NOTE — Some **controllers** may have difficulty reading a 16 bit unsigned integer into their internal format.

11.6.1.2 Reading the Parallel Poll Enable Register

The contents of the Parallel Poll Enable Register can be queried by sending *PRE?. The **device** responds with an <NR1 NUMERIC RESPONSE DATA>. element, see 10.24.

11.6.1.3 Writing the Parallel Poll Enable Register

The Parallel Poll Enable Register is written to by sending the *PRE command followed by a number in <DECIMAL NUMERIC PROGRAM DATA> format to the **device**. The number is converted to binary and placed in the register, see 10.23.

11.6.1.4 Clearing the Parallel Poll Enable Register

The Parallel Poll Enable Register is cleared by

1) Setting the value of the register to zero
2) Performing a power-on, if the power-on-status-clear flag is TRUE or if *PSC is not implemented

11.6.2 Reading ist Without a Parallel Poll

The state of ist can be queried by sending the *IST?common query. The **device** shall return either a “0,” if ist is FALSE, or a “1,” if ist is TRUE. The number returned shall be an <NR1 NUMERIC RESPONSE DATA> element, see 10.15.

12. Device/Controller Synchronization Techniques

12.1 Overview

This section describes techniques that may be used to ensure synchronization between a **device** and a **controller**. Three basic techniques are shown:

1) Force sequential execution
2) Wait for response in **device's** Output Queue.
3) Wait for a Service Request.

Detailed examples of these techniques appear in Appendix B.

One potential problem with commands that take appreciable time to finish is that the application program needs to know when they have finished. Consider the case of a unit under test connected to a bus-controllable power supply and a bus-controllable digital voltmeter (DVM). The application program needs to know that the power supply output has reached the desired voltage setting before it commands the DVM to take a measurement. This confirmation that the output is valid is an illustration of what is meant by **device/controller** synchronization.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

12.2 Sequential and Overlapped Commands

Device commands fall into two broad classes. The first is the class of "Sequential Commands." A command is a Sequential Command when Execution Control waits for the resulting **device** action to complete before it initiates the next **device** action, see 6.1.7.

The other class of commands is "Overlapped Commands." An Overlapped Command is a command that allows execution of subsequent commands while the **device** operations initiated by that Overlapped Command are still in progress.

The **device** designer chooses whether a device-specific command is Sequential or Overlapped. Refer to 12.8.1 for further information. The designer may also choose to implement device-specific commands that change a command's type from Sequential to Overlapped and vice versa.

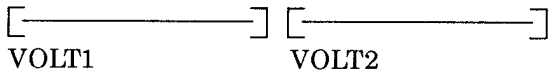
Device documentation shall specify for each command whether it is an overlapped or a sequential command.

12.2.1 Illustration of Sequential CommAnds

As an illustration, consider the case of a bus-controllable power supply with two outputs. The command VOLT1 NRf sets the voltage of one output and VOLT2 NRf sets the voltage of the other. If the VOLT commands are Sequential Commands, then the <PROGRAM MESSAGE>

VOLT1 1.23; VOLT2 1.00 <PMT>

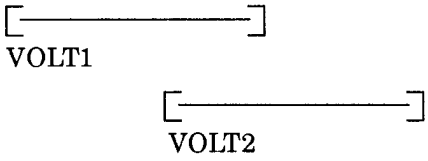
(where <PMT> stands for the <PROGRAM MESSAGE TERMINATOR>) would set the first output to 1.23 V, and then after that was done, would set the second output to 1.00 V. In a timeline, the commands appear as



The symbol "[" stands for initial. ion of the associated **device** operations and "]" stands for the completion of those operations. Note that completion of **device** operations is not the same as completion of execution of the command. See 6.1.7 for the distinction.

12.2.2 Illustration of Overlapped Commands

If the VOLT commands are Overlapped Commands, the VOLT2 command would begin execution before the **device** operations initiated by the VOLT1 command have completed. The timeline is



Some method is needed to show that all commands have finished. Sequential Commands are easy because the next command is not executed until the previous Sequential Command is finished. Overlapped Commands are more difficult because **device** operations that were initiated several commands ago may still not be finished.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

12.3 Pending-Operation Flag

The Pending-Operation flag indicates that all operations started by an Overlapped Command are completed and their result is valid or that the operations were aborted.

Each Overlapped Command shall have a corresponding Pending-Operation flag. A Pending-Operation flag is set TRUE when the corresponding command is sent by the Execution Control block, see 6.1.7, to the Device Function block, see 6.1.8. The flag is set FALSE when all the **device** operations initiated by that command have been completed or aborted (for example, by execution of the *RST command).

For instance, the operation of changing the output voltage of a power supply is complete only when the output is known to have reached the programmed voltage within manufacturer specified tolerance. The operation of changing the range on a DVM is complete only when the DVM is ready to make a measurement in that range. Fig 12-1 describes control of these flags.

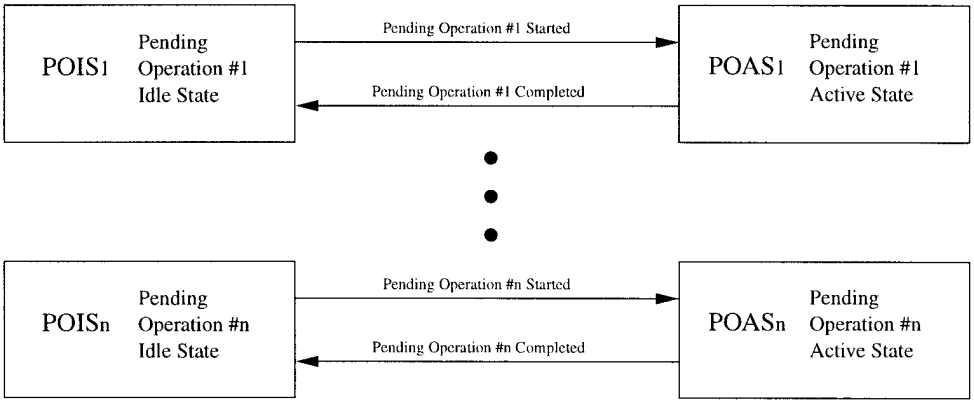


Figure 12-1—Pending Operation Diagram of Selected Overlapped Commands

12.4 No-Operation-Pending Flag

The No-Operation-Pending flag is used to indicate whether the **device** has completed operations initiated by selected Overlapped Commands. The **device** designer may choose to have the No-Operation-Pending flag report the status of all or a subset of all Overlapped Commands. The **device** designer may provide device-specific commands to select which Overlapped Commands shall have their status reported by the No-Operation-Pending flag.

The Pending-Operation flags associated with the selected Overlapped Commands shall be ORed together, and the result shall be inverted to generate the No-Operation-Pending flag. The No-Operation-Pending flag is TRUE if the **device** has no selected Overlapped Commands.

12.5 Controller/Device Synchronization Commands

The *OPC and *WAI commands and the *OPC? query allow the application programmer to maintain **controller/device** synchronization.

12.5.1 The *WAI Common Command

The *WAI command shall prevent the **device** from executing any further commands or queries until the No-Operation-Pending flag is TRUE. No special provision is made for Sequential Commands because any preceding Sequential Command will complete execution before the Parser hands the next command (including the *WAI command) to the Execution Control block.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

12.5.1.1 A Sample Program Message Without the *WAI Command

An illustration of the use of the *WAI command is the following: Consider a data-logging **device** that is commanded to take a measurement with the device-specific command START and to read the present time by TIME?. The START command is an Overlapped Command and takes appreciable time to perform. The <TERMINATED PROGRAM MESSAGE>

START; TIME? <PMT>

causes the **device** to take a measurement and return the time at which the measurement was begun. The timeline and flags are shown in Fig 12-2.

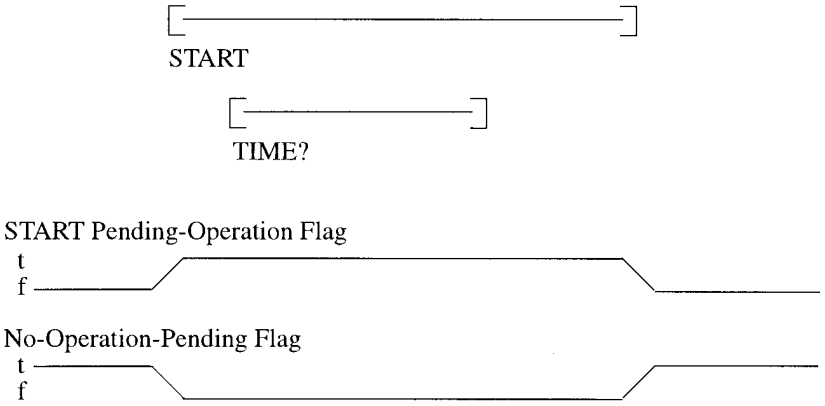


Figure 12-2—Timing Diagram — Sample Program Message Not Using *WAI Command

12.5.1.2 A Sample Program Message With the *WAI Command

The <TERMINATED PROGRAM MESSAGE>

START; *WAI; TIME? <PMT>

forces the TIME? query response to be the time at which the measurement was finished. Thus, the *WAI command allows the application programmer to force sequential execution of Overlapped Commands. The timeline and flags are shown in Fig 12-3.

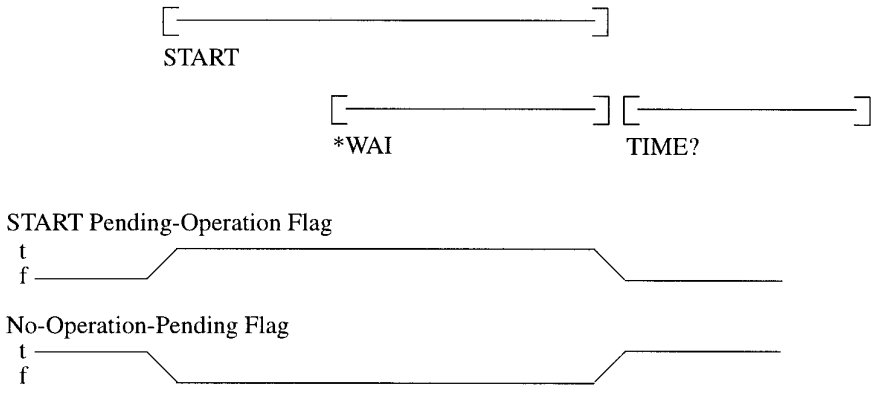


Figure 12-3—Timing Diagram — Sample Program Message Using *WAI Command

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Detailed illustrations of the use of the *WAI command appear in Appendix B.

The *WAI command's holdoff action shall be canceled by power-on and by the **dcas** message. Since the *WAI command holds off the **device** execution control until the No-Operation-Pending flag is TRUE, the commands *RST and *CLS can have no effect on the operation of *WAI.

12.5.2 The *OPC Common Command

The *OPC command allows synchronization between a **controller** and several **devices.** The OPC event-bit in the Standard Event Status Register (ESR) is used to effect the synchronization. Detailed illustrations of the use of the *OPC command appear in Appendix B.

12.5.2.1 The Operation Complete Command Diagram

The *OPC command shall perform as described in Fig 12-4 and the descriptions given throughout 12.5.2.

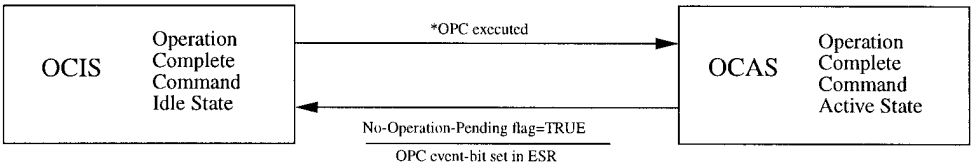


Figure 12-4—Operation Complete Command Diagram

12.5.2.1.1 Operation Complete Command Idle State (OCIS)

The following conditions place the **device** into OCIS:

1) Power-on, see 5.12.
2) Receipt of the **dcas** message, see 5.8.
3) *CLS is executed, see 10.3.
4) *RST is executed, see 10.32.

The **device** shall exit OCIS and enter Operation Complete Command Active State (OCAS) when the *OPC command is executed.

12.5.2.1.2 Operation Complete Command Active State (OCAS)

In OCAS, the **device** continuously senses the No-Operation-Pending flag. When the No-Operation-Pending flag is sensed TRUE, the **device** shall set the OPC event-bit in the Standard Event Status Register (ESR), exit OCAS, and enter Operation Complete Command Idle State (OCIS).

12.5.2.2 Sample Use of the *OPC Command

The previous example in 12.1 of the power supply, DVM, and **device** under test illustrates the effect of *OPC. The problem is to program the DVM to take a measurement only after the power supply has finished changing its output voltage.

The application programmer may enable bit 0 (corresponding to the OPC event-bit) in the Standard Event Status Enable Register and also bit 5 (corresponding to the ESB summary-message) in the Service Request Enable Register so that the **device** will request service whenever the OPC event-bit becomes TRUE. A <TERMINATED PROGRAM MESSAGE> of

*CLS; VOLT 5.20; *OPC <PMT>

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

then commands the power supply to clear its Standard Event Status Register and Status Byte, set its output voltage to 5.20 V, and then request service once the output voltage is valid. After the service request is detected, the application program can then command the DVM to take a measurement. The timeline and flags are shown in Fig 12-5.

12.5.3 The *OPC? Common Query

The *OPC? query allows synchronization between a **controller** and a **device** using the MAV bit in the Status Byte or a read of the Output Queue. Detailed illustrations of the use of this function appear in Appendix B. Note that, unlike the *OPC command described in 12.5.2, the *OPC? query does not in any way affect the OPC Event bit in the Standard Event Status Register (ESR).

12.5.3.1 The *OPC? Query Diagram

The *OPC? query shall perform as described in Fig 12-6 and the descriptions given throughout 12.5.3.

12.5.3.1.1 Operation Complete Query Idle State (OQIS)

The following conditions place the **device** in OQIS:

1) Power-on, see 5.12.
2) Receipt of the **dcas** message, see 5.8.
3) *CLS is executed, see 10.3.
4) *RST is executed, see 10.32.

The **device** shall exit OQIS and enter Operation Complete Query Active State (OQAS) when the *OPC? query is executed. See Fig 12-6.

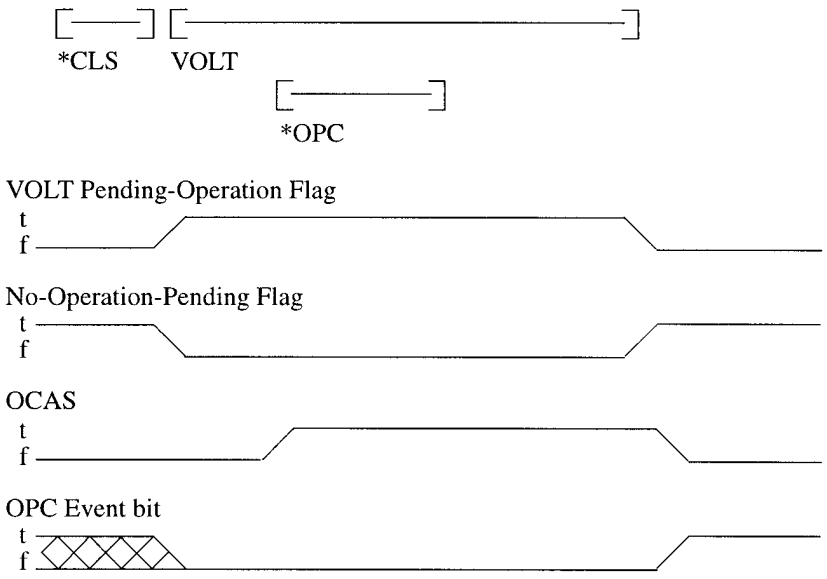


Figure 12-5—Timing Diagram — Sample Program Message Using *OPC Command

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

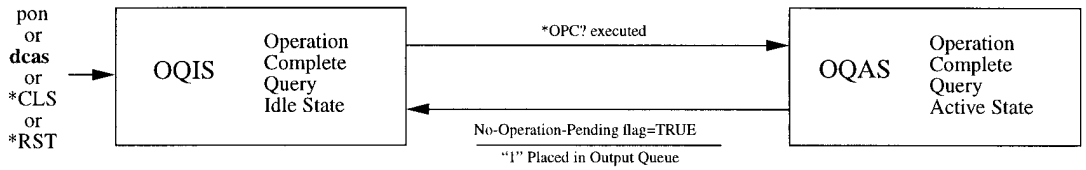


Figure 12-6—Operation Complete Query Diagram

12.5.3.1.2 Operation Complete Query Active State (OQAS)

In OQAS, the **device** continuously senses the No-Operation-Pending flag. When the No-Operation-Pending flag is sensed TRUE, the **device** shall place an ASCII character "1" in the Output Queue. A consequence of this action is that the MAV bit in the Status Byte will become TRUE. The **device** shall also exit OQAS and enter OQIS.

12.5.3.2 Sample Use of the *OPC? Query — No Service Request

The following technique of using the *OPC? query avoids the use of service requests and serial polls. Bit 4 (corresponding to the MAV summary-message) in the Service Request Enable Register is set to a value of zero (disabled). The<TERMINATED PROGRAM MESSAGE>

VOLT 5.20; *OPC? <PMT>

is then sent to the power supply. The application program then attempts to read the *OPC? query response from the power supply. The **device** will not put a response to the *OPC? into the Output Queue until the voltage is valid. After the response is received, the **controller** commands the DVM to take a measurement.

12.5.3.3 Sample Use of the *OPC? Query — Service Request Method

The following technique makes use of service request and serial poll. Bit 4 (corresponding to the MAV summary-message) in the Service Request Enable register is set to a value of one (enabled) and the <TERMINATED PROGRAM MESSAGE>

VOLT 5.20; *OPC? <PMT>

is sent to the power supply. The power supply requests service when the output voltage is valid. Then the DVM is commanded to take a measurement.

While reading the response to the *OPC? query removes the complication of dealing with service requests and the attendant serial poll, it has the penalty that both the **system bus** and the **controller** handshake are in a temporary holdoff state while the **controller** is waiting to read the *OPC? query response. The method using the *OPC command and the OPC event-bit in the Standard Event Status Register or the method using the *OPC? query and the MAV summary-message in the Status Byte Register both require the use of a serial poll. However, they allow both the **system bus** and the **controller** to perform other operations while waiting for the service request.

12.6 Synchronization With External-Control-Signals

The techniques described in this section may be used to provide synchronization between external-control-signals, as defined in 5.6.1.4, and the application program. The **device** designer may provide device-specific Overlapped Commands that depend upon receipt of external-control-signals. A **device** operation may be initiated by the command and completed as a consequence of receipt of an external-control-signal.

For example, consider a DVM that has the command "TRG:EXT" to select an external trigger and the Overlapped Command (and query) "MEAS?" to take a measurement when triggered. The DVM will make the Pending-Operation flag TRUE upon receipt of the following <TERMINATED PROGRAM MESSAGE>:

*CLS; TRG:EXT; MEAS? <PMT>

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

After the external trigger signal is applied, the measurement is taken, and then the Pending-Operation flag is made FALSE.

All the synchronization methods using the *WAI, *OPC commands, and the *OPC? query may be used with Overlapped Commands that use external-control-signals.

12.7 Improper Usage of *OPC and *OPC?

The *OPC command and *OPC? query are intended to appear as the last <PROGRAM MESSAGE UNIT> in a <PROGRAM MESSAGE>. The result of the <TERMINATED PROGRAM MESSAGE>

*CLS; OL1; OL2; *OPC; OL3 <PMT>

will depend upon the relative timing of the completion of the **device** operations started by the “OL1” and “OL2” Overlapped Commands and the execution of the “OL3” Overlapped Command.

The No-Operation-Pending flag may still be FALSE when the Overlapped Command “OL3” is executed. In this case, the OPC event-bit becomes TRUE when the Overlapped Command “OL3” is finished because the No-Operation-Pending flag does not go TRUE before the Pending-Operation flag corresponding to the “OL3” command goes TRUE. The timeline and flags for this case are as shown in Fig 12-7.

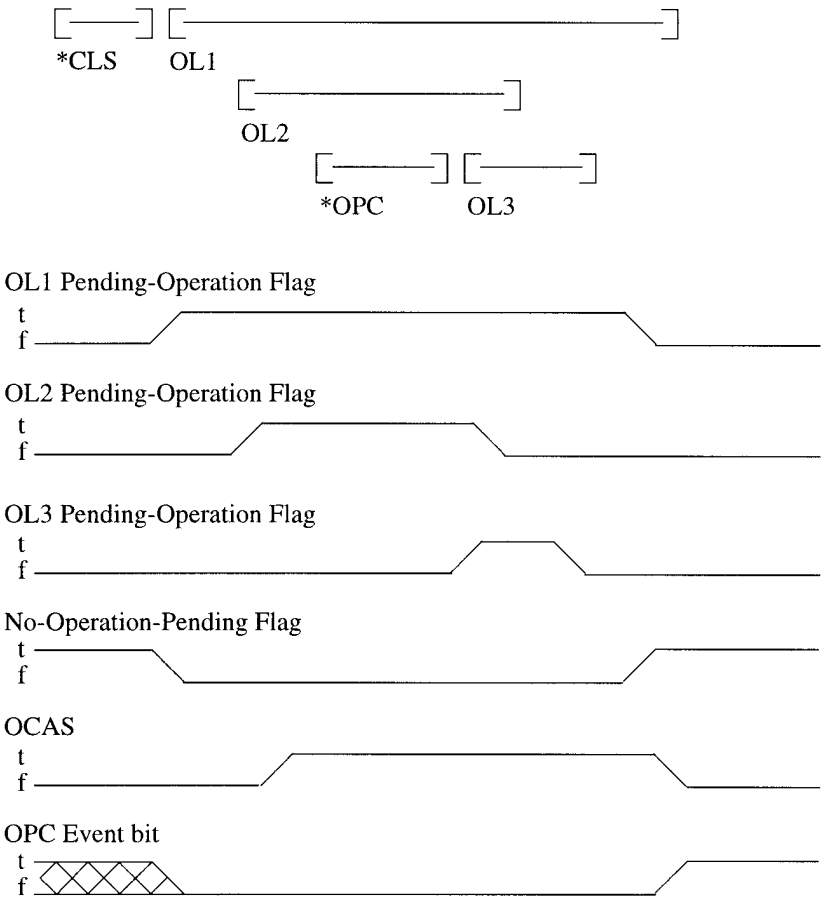


Figure 12-7—Timing Diagram — Improper Operation Example 1

However, if the No-Operation-Pending flag becomes TRUE before the Overlapped Command “OL3” is executed, then the OPC event-bit will already be TRUE. The timeline and flags for this case are as shown in Fig 12-8.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

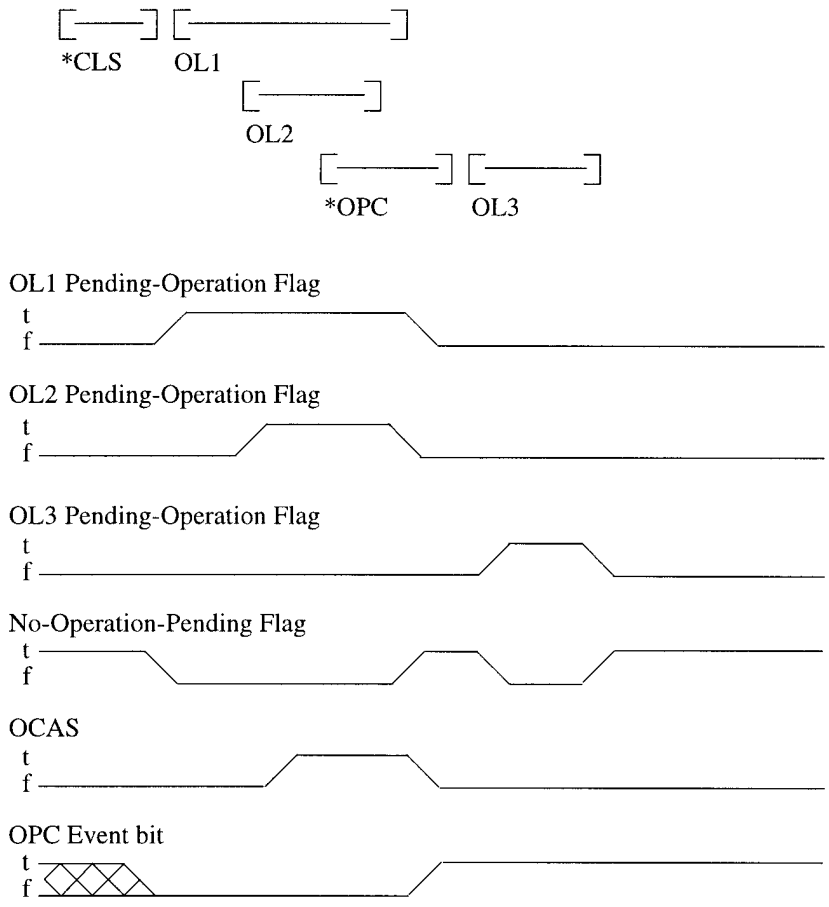
12.8 Design Considerations

This section is intended to guide the **device** designer in choosing which commands should be Overlapped Commands and which should be Sequential commands. It also provides guidance to the **device** designer about how the No-Operation-Pending flag shall be generated.

12.8.1 Overlapped Commands

In general, a **device** designer will want to make commands sequential if they complete in a short time and overlapped if they take a long time to complete. “Short time” and “long time” must be considered in the context of other **device** commands.

For example, consider a data logger that can back up its data to tape without interrupting the logging of new data. This back up takes I h to complete, so the backup operation should be overlapped. Assume that logging new data takes 1 min. This is a long time compared to many of the **device** operations (for example, responding to *STB? query), but a short time compared to the backup operation.



NOTE: If OL1 or OL2 operation is extended beyond *OPC, then OL3 will decide the time at which the OPC event occurs.

Figure 12-8—Timing Diagram — Improper Operation Example 2

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Consider the case in which the **device** designer chooses to make the data-logging operation overlapped. When the application program commands the data logger to take a measurement, the No-Operation-Pending flag will go FALSE for 1 min and then TRUE. If the application program commands the data logger to take a measurement while a backup operation is in progress, the No-Operation-Pending flag will remain FALSE until the backup completes, possibly as long as 1 h. During this time, the application program has no way of knowing whether the measurement is complete.

Now consider the case in which the **device** designer chooses to make the data logging operation sequential. This may be an acceptable solution since the "MEASURE" command by itself will not cause a delay. If the **controller** attempts to send additional commands before the data logging operation is complete, then the **controller** and bus will be "stalled" until the data logger is ready to execute the new commands. Note that this method is very sensitive to **device** implementation. If the **device** has no Input Buffer, sending the <TERMINATED PROGRAM MESSAGE>

MEASURE; <space> <PMT>

could cause the **device** to wait after the semicolon until the measurement completes and it is ready to accept the rest of the <TERMINATED PROGRAM MESSAGE>. A "reasonable length" input buffer, see 6.1.5, would be large enough to hold the command, extra whitespace, and <PROGRAM MESSAGE TERMINATOR> so that the bus handshake would not be stalled during the measurement.

If the hypothetical data logger had to return some data to the **controller** at the end of each measurement, sequential operation would not be acceptable since the **controller** and bus would be stalled for the duration of each measurement. In this case, the **device** designer might choose to ignore the "BACKUP" Pending-Operation flag when determining the state of the No-Operation-Pending flag. (The "device-defined subset" in 12.4 would not include the "BACKUP" Pending-Operation flag.) This would allow the application programmer to use the standard *WAI, *OPC, and *OPC? commands to synchronize measurements. The **device** designer would have to provide some other device-specific means of determining when the backup operation had completed.

The **device** designer could also create a device-specific command to select which pending-operation flags would affect the No-Operation-Pending flag. This would allow the application programmer to use the *WAI, *OPC, and *OPC? commands to wait for the measurement to complete, for the backup to complete, or both.

12.8.2 Execution Error Handling

If a <PROGRAM MESSAGE> causes an Execution Error, the associated operation is aborted, the Pending-Operation flag is made FALSE, and the **device** reports an Execution Error.

12.8.3 Operation Complete

The **device** designer shall determine the point in time when Overlapped Commands complete their associated **device** operations. The following principles shall be followed in determining when to set the Pending-Operation flag FALSE for each associated device-defined Overlapped Command.

The **device** designer must realize the importance of the requirements of this section. These requirements allow application software to determine when external components of a **system** may utilize the results of a **device** operation. Thus, the **device** shall not prematurely report that an operation is complete. The **device** may indicate that the operation is complete anytime after the operation really is complete.

The **device** designer may include special hardware, such as sensors, for the **device** to determine when an operation is complete. Alternatively, the **device** designer may design the **device** to wait for an appropriate amount of time to elapse. Any method is acceptable as long as the results of an operation are valid whenever the **device** reports that operation to be complete. A **device** that reports an operation to be complete when, in fact, it is not, is either misapplied, out of calibration, or defective. A signal generator connected to a short circuit is an example of a misapplication. For each command, the **device** documentation shall specify the functional criteria that are met when an operation complete message is generated in response to that command.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The criteria that shall be satisfied before an operation is reported complete are shown separately for two classes of **devices:** stimulus-devices, see 12.8.3.1, and response-devices, see 12.8.3.2. A **device** may contain functions of both types.

12.8.3.1 Stimulus-Devices

A stimulus-device generates a signal that is used to stimulate an external piece of equipment in a **system.** An example of a stimulus-device is a power supply.

When a device-specific Overlapped Command causes the **device** to generate a signal, the **device** shall not indicate that the associated pending operation is complete until one of the following has occurred:

1) The signal on its output port is in a valid state.
2) The operation has been aborted.

The **device** may provide device-specific commands that allow the application programmer to select which characteristics of the signal applied at the **device's** output port will set the Pending-Operation flag FALSE. For example, these commands might select the accuracy to which an output signal must settle before the **device** indicates that an operation is complete.

12.8.3.2 Response-Devices

A response-device is a **device** that responds to an externally provided signal. An example of a response-device is a DVM that measures an external signal.

When a device-specific Overlapped Command causes the **device** to respond to an external signal, the **device** shall not indicate that the associated pending operation is complete until one of the following has occurred:

1) The response is complete.
2) The operation has been aborted.
3) The **device** is ready for the **controller** to begin a read operation in response to a query Overlapped Command.

The **device** may provide device-specific commands that allow the application programmer to select which characteristic(s) of the response sets the Pending-Operation flag FALSE. For example, these commands might select the degree of accuracy to which the signal is measured.

13. Automatic System Configuration

13.1 Introduction

When a test **system** is configured or reconfigured, addresses must be assigned to the **devices** so that the **controller** application program can send messages to and receive messages from the desired **devices.** This section provides tools in the form of protocols, commands, and guidelines to automate this task for **devices** independent of compliance with IEEE 488.2 . It also discusses how the information generated by the tools can be used by an application program to automatically adjust to test **system** hardware changes.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

13.2 Overview

The following protocol and common command combinations support automatic configuration:

Controller Protocol	Common Command	Purpose of Combination
Find Listeners (see 17.6)	Disable Listener (*DLF) (see 10.6 and 13.4.1)	To find non-address-configurable IEEE 488.1 and IEEE 488.2 **devices** (*AAD/*DLF not implemented)
Set Address (see 17.7)	Accept Address (*AAD) (see 10.1 and 13.4.2)	To find address-configurable IEEE 488.2 **devices** (*AAD/*DLF implemented)

An overview of the use of these combinations is given below and does not represent an actual implementation. Fig 13-1 illustrates a comprehensive search of **devices** on an IEEE 488.1 bus.

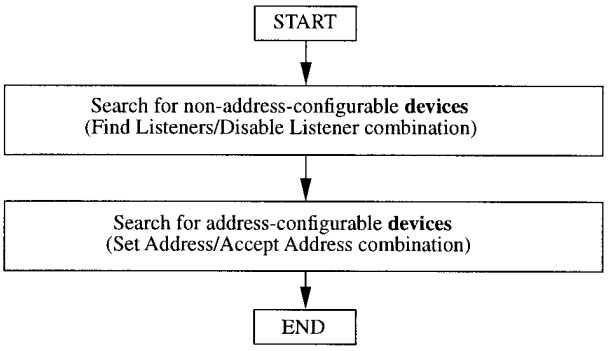


Figure 13-1—Comprehensive Device Search

The Non-Address-Configurable Device Search, in turn, is comprised of the elements shown in Fig 13-2 followed by the Address-Configurable Device Search elements in Fig 13-3.

The Find Listeners **controller** protocol mentioned in Fig 13-2 and Set Address in Fig 13-3 are further described in Figs 13-4 and 13-5, respectively. Set Address protocol interacts with the Accept Address common command of Fig 13-6 to implement the Address-Configurable Device Search.

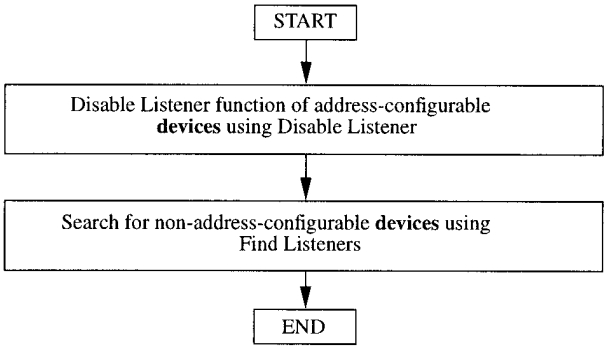


Figure 13-2—Non-Address-Configurable Device Search

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

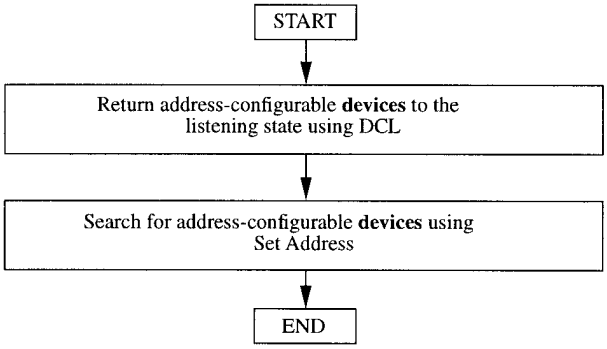


Figure 13-3—Address-Configurable Device Search

The Set Address **controller** protocol and Accept Address common command interact as follows:

The Present-Identifier-Byte procedure in each **device** endeavors to guide the **controller** to its current byte, which is detected by the Acquire-Byte procedure in the **controller.** The Acquire-Byte procedure acquires each byte via a binary search until one entire identifier is acquired, see Appendix C. At this. point, only one **device** is left participating, and the **controller's** Configure-Device procedure assigns the **device** an address that is accepted by the **device's** Present-Device-Identifier procedure. This scenario repeats until all **devices** have been detected and have been assigned addresses.

13.3 Generic Approach to Automatic System Configuration

13.3.1 Address Assignment

The method of address assignment described below is a generic approach and may be tailored to specific requirements. The non-address-configurable **devices** are detected first, followed by address-configurable **devices.** The application program is responsible for supplying a list of addresses to be searched for the presence of **devices** and a table describing the desired final configuration.

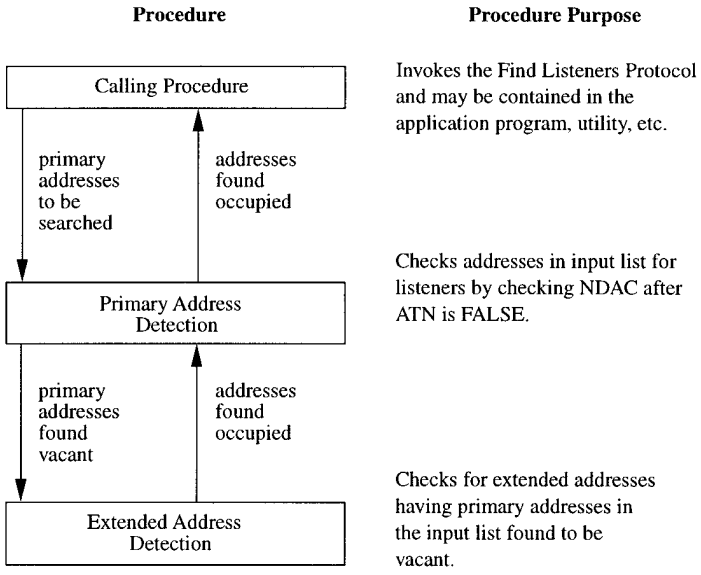


Figure 13-4—Find Listeners Protocol

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13.3.1.1 Types of Devices

Three types of devices may reside on an IEEE 488.1 bus:

1) Non-IEEE 488.2 **devices**
2) Non-address-configurable **devices** (no *AAD)
3) Address-configurable **devices** (implement *AAD)

The first two types are defined as non-address-configurable. The third type is defined as address-configurable.

13.3.1.2 Non-Address-Configurable Device Detection

Non-address-configurable **devices** are detected by an address probing routine (**controller** Find Listeners protocol, see 17.6) that searches each address specified in the input address list for a listener **device** or **devices.** During the poll, the listener capabilities of the address-configurable **devices** are disabled by the *DLF common command (Disable Listener Function, see 10.6) to prevent them from responding to the poll. The protocol performs the search by noting if the NDAC line is TRUE after the address is sent. If so, a **device(s)** exists. Otherwise, the address location is empty. If the address location is occupied, the address is stored in a non-address-configurable address table.

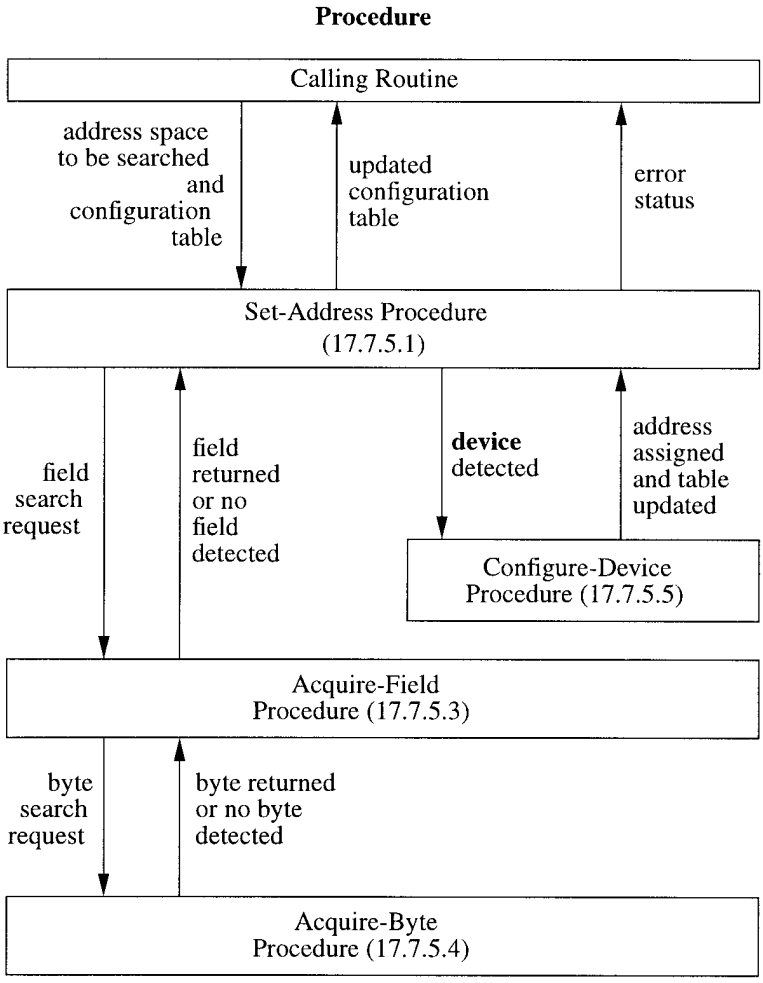
13.3.1.3 Address-Configurable Device Detection

Address-configurable **devices** are detected via a binary searching routine (**controller** Set Address protocol, see 17.7.2, and the *AAD Common Command). Each **device** seeks detection by guiding the **controller** to its identifier with its Listener and Service Request functions. The identifier is made up of the first three fields of the *IDN? query response (manufacturer, model number, and serial number) and the **device's** current listen address. Once an identifier is detected, the **controller** assigns an address to the associated **device** according to a user-supplied configuration table. The table will normally exclude addresses occupied by the non-address-configurable **devices** to prevent communication conflicts.

--`,,```,,,,--`-`,,`,,`,`,,`---

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



PROCEDURE	PROCEDURE PURPOSE
Calling Routine	Invokes the Set Address Protocol and may be contained in the application program, or a utility, etc.
Set-Address Procedure (17.7.5.1)	Searches for address-configurable **devices** in the input address space, manages the acquisition of **device** identifier fields, see 13.4.2.2, and configures the **device**.
Configure-Device Procedure (17.7.5.5)	Updates configuration table to include the **device** detected and assigns an address to that **device**.
Acquire-Field Procedure (17.7.5.3)	Manages the acquisition of bytes within a **device** identifier field, see 13.4.2.2.
Acquire-Byte Procedure (17.7.5.4)	Performs a binary search to identify the lowest active **device** identifier data byte being presented by a **device(s)** and instructs only that **device** to continue participating.

Figure 13-5—Set Address Protocol

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

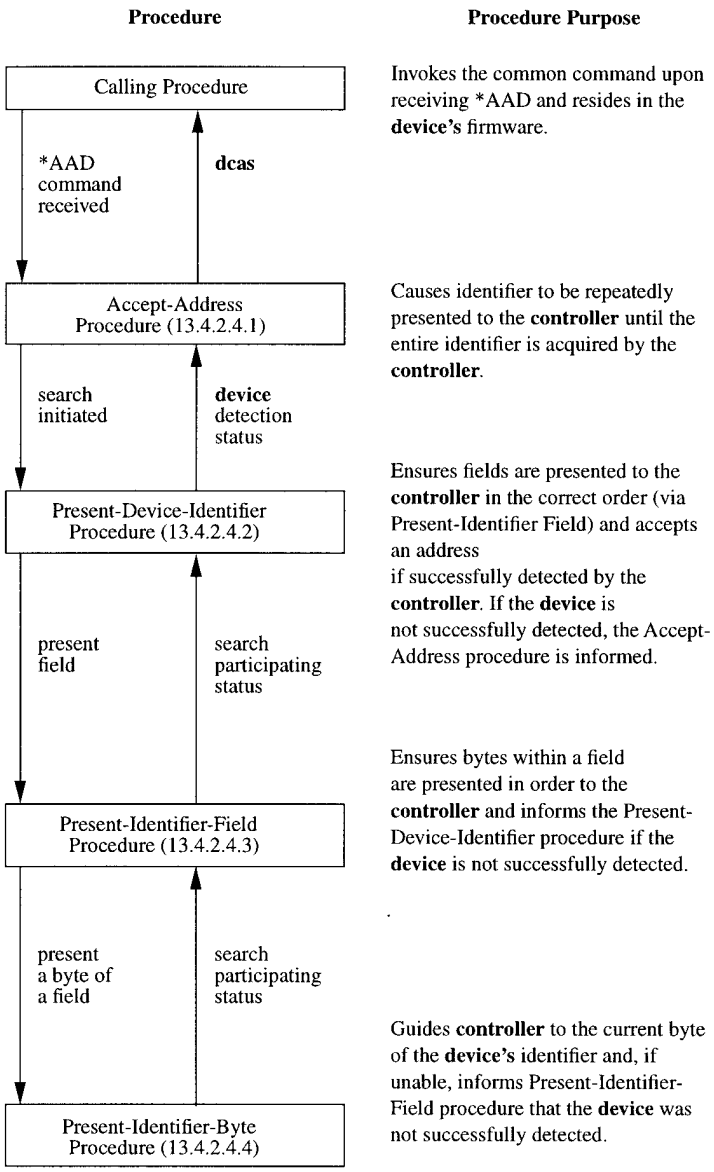


Figure 13-6—Accept-Address Common Command

13.3.1.4 System Configuration Table

After all **devices** have been detected, the application program may create a **system** configuration table that lists each of the **devices** detected by these protocols. This application program may also place other data into the table, such as that returned by the *RDT? or *PUD? common queries. A number of the **controller** protocols described in Section 17. require an input list of addresses. The application program may use the **system** configuration table to provide these addresses to the protocols. Thus, the application software can automatically adjust itself in response to any configuration or reconfiguration of the test **system** that it controls.

13.3.1.5 Limitations

The Find Listeners protocol cannot detect multiple **devices** at the same address. Thus, the user is responsible for ensuring that each non-address-configurable **device** resides at a unique address. The protocol also will not work if

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

there are **devices** in the **system** that take more than 1 ms to cease listening after the **controller** sends ATN FALSE when they are not addressed.

The Set Address protocol will unambiguously detect every address-configurable **device** regardless of its address, except when an identifier is not unique. Such a situation can occur if two like **devices** have "0" for a serial number and their addresses are the same at the start of the search. If the user changes one of these addresses, then the searching method will operate correctly.

The Set Address protocol will not work if any non-address-configurable **device** in the **system** asserts SRQ TRUE during the Set Address protocol. In some cases, serial polling all **devices** found by the Find Listener protocol before starting the Set Address Protocol may be sufficient to clear the SRQ line. In other cases, some other means must be used to make these **devices** set SRQ FALSE for the duration of the protocol.

13.3.1.6 Procedure

1) Send all primary and extended listen addresses, which causes all **devices** in the **system** to listen.
2) Send **dcas** message *DLF <NL^END> and wait 100 ms, which disables all address-configurable **devices** from listening.
3) Send *CLS <NL^END>. This command clears the Query Error in all non-address-configurable **devices** caused by receiving the unrecognized command, *DLF, and causes them to release SRQ.
4) Perform Find Listeners protocol at all addresses. Each address is tested for the presence of non-address-configurable **devices.** Afterwards, all of those **devices** are unlistened.
5) Place the non-address-configurable **devices** (found in the previous step) into the **system** configuration table.
6) Send DCL and wait 100 ms, which restores address-configurable **devices** to normal listening operation (LACS or LADS).
7) Perform the Set Address protocol on all currently listen-addressed **devices.** This protocol identifies all address-configurable **devices,** but does not change their current addresses.
8) Place address-configurable **devices** into the **system** configuration table.
9) Perform the Set Address protocol again to assign new addresses to address-configurable **devices.**

The application program should include only **devices** that need to be assigned new addresses in this step. (If the application program does not need to assign specific addresses to specific **devices,** this step may be omitted.) At the conclusion of this procedure, the **system** configuration table contains a list of

1) Address-configurable **devices,** including their manufacturer, model number, serial number, and address
2) Non-address-configurable **device** addresses

13.3.2 Device Identification

The manufacturer and model number of a **device** may be used by an application to assign a type of **device** (for example, DVM, power supply) to a particular bus address. In some applications, keeping a record of which specific pieces of test equipment were used for a given test is important. For example, this information can be used to maintain a calibration history of a **device.**

13.3.2.1 Addition of Non-IEEE 488.2 Devices

At the conclusion of the previous procedure, the **system** configuration table includes the manufacturer, model number, serial number, and address for each address-configurable **device.** The table contains only the address for each non-address-configurable **device.** The application must obtain the remaining three pieces of information for non-IEEE 488.2 **devices** from the keyboard, a disk file, or other source. Non-IEEE 488.2 **devices** must be identified in the **system** configuration table so that they can be excluded from IEEE 488.2 -specific procedures.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13.3.2.2 Addition of IEEE 488.2 Non-Address-Configurable Devices

After the non-IEEE 488.2 **devices** are identified, the application may use the "*IDN?" query to fill in the rest of the table entries for non-address-configurable **devices.**

13.3.2.3 Additional Configuration Commands

A number of common commands and queries are available to provide information useful within an automatic test **system.** These commands include Protected User Data (*PUD and *PUD?), Resource Description Transfer (*RDT and *RDT?), Calibration (*CAL?), and Option Identification Query (*OPT?).

Because these and other related common commands are optional, the application program must be able to determine if a particular **device** has implemented a given command or query. The application may derive this information from its own knowledge of a **device's** capabilities from a Resource Description. Another way is to send the command or query, and use the **device's** required status reporting facilities to determine if the **device** does not implement the command. For example, the application program could determine if the **device** implements the *RDT? common query with the following procedure:

NOTE — Quotes around data strings are not part of the data sent.

1) Send "*ESE 1;*SRE 32 ;*CLS<NL^END>." This <TERMINATED PROGRAM MESSAGE> enables reporting of the OPC event (only) and clears the status registers.
2) Send "*RDT?;*OPC<NL^END>." The **device** may or may not implement the *RDT? query, but is required to implement the operation complete (*OPC) command. The **device** will set the OPC event bit in the Standard Event Status Register when it executes the *OPC command. (An assumption is made that the **device** has no pending operations, which is a reasonable assumption during **system** initialization.) The **device** will eventually set the Event Status Bit (ESB, bit 5) in the **device's** Status Byte Register and cause a service request.
3) The application may wait for the service request signal or may repeatedly poll the **device** until the **device's** status byte indicates a service request (RQS TRUE). (The application should not assume that a service request is caused by this **device,** but should always check the **device** status byte before continuing.)
4) Send "*ESR?<NL^END>" without attempting to read the *RDT? query response. Note that this sequence is a violation of the normal **device** message exchange protocol, see Section 6., and may cause the **device** to indicate a Query Error, which the application program is choosing to ignore.
5) Read the *ESR? response. The Operation Complete (OPC, bit 0) event bit should be TRUE (value of 1). The Command Error (CME, bit 5) event bit will be FALSE (value of 0) if the **device** does implement the *RDT? common query and TRUE (value of 1) if the **device** does not. The application program should ignore the Query Error (QYE, bit 2) event bit, which may or may not be set.

13.4 Detailed Requirements of the Auto Configuration Commands

13.4.1 *DLF Common Command Requirements

The *DLF common command shall disable the **device's** normal Listener' function until a Device Clear (DCL) interface message, an interface clear (IFC) interface message, or power-on (pon) is received. The **device** shall enter Listener Idle State (LIDS) within 100 ms after the **device's** acceptor handshake enters Acceptor Wait for New Cycle State (AWNS) on the last byte of the following <TERMINATED PROGRAM MESSAGE>:

*DLF<NL^END>

where "DLF" may be in any mixture of uppercase or lowercase. This timing requirement does not apply if

1) This is not the first <PROGRAM MESSAGE UNIT> received after the **dcas** message.
2) Any other syntactic elements are sent with the *DLF command before the <PROGRAM MESSAGE TERMINATOR>.
3) Any <PROGRAM MESSAGE TERMINATOR> other than <NL^END> is used.

--`,,`,,,-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

If the **device** receives a *DLF command that is not the first <PROGRAM MESSAGE UNIT> received after the **dcas** message, or is not the only <PROGRAM MESSAGE UNIT> in a <PROGRAM MESSAGE>, it shall indicate an Execution Error and shall not execute the *DLF command.

The **device** shall not indicate an error solely because the *DLF is preceded or followed by whitespace or because a <PROGRAM MESSAGE TERMINATOR> other than <NL^END> is used; but it may not meet the timing requirement.

After receiving *DLF, the Listener function shall remain disabled until the **device** receives a **dcas** message. During this time, the Talker function, status reporting mechanism, and Controller function (if present) may continue normal operation or may be disabled. The **device** shall return to normal operation with the Listener function in Listener Addressed State (LADS) or Listener Active State (LACS) within 100 ms after receiving the **dcas** message. The **device** shall also return to normal operation in Listener Idle State (LIDS) following power-on or the Interface Clear (IFC) interface message, but shall not return to normal operation because of any other IEEE 488.1 bus condition.

13.4.2 *AAD Common Command Requirements

The *AAD common command prepares the **device** to participate in the controller's Set Address protocol. The **device's** normal input buffering, command processing, and status reporting mechanisms are disabled during the protocol.

13.4.2.1 Device Protocol Initiation

The **device** shall be ready to participate in the Set Address protocol within 100 ms after the **device's** Acceptor Handshake function enters AWNS on the last byte of the following <TERMINATED PROGRAM MESSAGE>:

*AAD<NL^END>

where "AAD" may be in any mixture of uppercase or lowercase. This timing requirement does not apply if

1) This is not the first <PROGRAM MESSAGE UNIT> received after the **dcas** message.
2) Any other syntactic elements are sent with the *AAD command before the <PROGRAM MESSAGE TERMINATOR>.
3) Any <PROGRAM MESSAGE TERMINATOR> other than <NL^END> is used.

If the **device** receives an *AAD command that is not the first <PROGRAM MESSAGE UNIT> received after the **dcas** message, or is not the only <PROGRAM MESSAGE UNIT> in a >PROGRAM MESSAGE>, it shall indicate an Execution Error and shall not execute the *AAD command. The **device** shall not indicate an error solely because the *AAD is preceded or followed by whitespace or a <PROGRAM MESSAGE TERMINATOR> other than <NL^END> is used; but it may not meet the timing requirement. The **device** shall return to normal operation at the end of the Set Address protocol or following power-on (pon), the Interface Clear (IFC) interface message, or the **dcas** message, but shall not return to normal operation because of any other IEEE. 488.1 bus condition. If a **device** receives a pon, IFC, or **dcas** prior to being assigned an address, it shall retain the address it possessed prior to the execution of this command.

13.4.2.2 Device Identifier

Each **device** that implements this command shall have an identifier made up of four fields:

1) Manufacturer
2) Model number
3) Serial number
4) Current address

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

For proper operation of this command, each byte of the identifier must have a decimal value greater than 31 and less than 127, and the identifier must be unique among all **devices** in the **system.** The first three fields of this identifier shall be identical to the first three fields of the Identification Query (*IDN?) query response. The fourth field shall consist of a single data byte (decimal value 32 through 62) that is identical to the **device's** current listen address and, if the **device** implements extended addressing, another data byte (decimal 96 through 126) that is identical to the **device's** current secondary address.

13.4.2.3 Protocol Operation

13.4.2.3.1 Overview

During the Set Address protocol, the **controller** attempts to identify each address-configurable **device** within the **system** by searching for a unique identifier. The **controller** identifies one **device** in each pass through the protocol. The search ordering is lexicographic by fields, with characters ordered according to their ASCII codes. If a field in one **device's** identifier is an initial substring of the same field in another **device's** identifier, the **device** with the longer field will be found first. During the pass, **devices** drop out of the search until only the first **device** (in the above ordering) is left. At the end of the pass, the **controller** assigns an address to that **device.** Once assigned an address, a **device** ignores subsequent bus traffic until all **devices** have been identified. The **controller** then restarts the search with the remaining **devices.** This process continues until all address-configurable **devices** have been identified and assigned new addresses.

13.4.2.3.2 SR Interface Function

The **controller** uses a binary search technique to find each byte in the identifier. The **device** uses its SR interface function to guide the **controller** through the search. The **device** shall send the rsv message FALSE if the value of the data byte presented by the **controller** is less than or equal to the value of the byte in the **device's** identifier. The **device** shall send the rsv message TRUE if the value of the data byte presented by the **controller** is greater than the value of the byte in the **device's** identifier. The **device** shall send the rsv message FALSE when the search has reached the end of an identifier field, or when the search has reached the end of the identifier.

13.4.2.3.3 Timing Requirements

The **device** shall send the correct value of the rsv message within i ms of the **device's** acceptor handshake entering AWNS. The **device** shall not make a transition from ANRS to ACRS (in preparation for the next byte) until the rsv message is stable.

13.4.2.3.4 Address Assignment

When the **device** has been identified, it shall accept one data byte from the **controller.** The value of this byte is the **device's** new listen address. The **device** shall set its primary address to the value of the lower five bits of this byte, and shall set its primary talk address so that the lower five bits of MLA and MTA are identical.

A **device** utilizing extended addressing shall accept an additional data byte from the **controller.** The **device** shall set its secondary address to the value of the lower five bits of this byte.

The **device's** Listener function shall not enter Listener Idle State (LIDS) when the new addresses are assigned, but shall remain in Listener Active State (LACS).

13.4.2.4 Algorithms

The following algorithms define a correct implementation of the *AAD command. The **device** shall implement these algorithms or a functionally equivalent procedure.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13.4.2.4.1 Accept-Address Procedure

Accept-Address watches for the STX character, ASCII character with the value 02 (2 decimal), that marks the start of a search pass. Present-Device-Identifier is called to present the **device's** identifier. If the **device** is not first in the search order, Present-Device-Identifier returns with the device-detected flag set FALSE. If the device-detected flag is FALSE, Accept-Address continues reading and discarding bytes until another STX character is received, indicating the start of a new search pass. If the device-detected flag is TRUE, this **device** has been identified. Accept-Address reads and discards all data until a **dcas** message restores normal operation.

```
BEGIN Accept-Address

  Disable input buffer
  Disable normal command processing
  Disable device status reporting
  Set rsv message FALSE

  Set device-detected to FALSE
  WHILE device-detected is FALSE
     BEGIN
       REPEAT
          Read data-byte
       UNTIL data-byte is ASCII STX (*decimal value of 2*)
       Perform Present-Device-Identifier
       (* Present-Device-Identifier will set device-detected TRUE if
        the controller has successfully identified this device *)
  END

  WHILE NOT in DCAS state
     Read and discard data bytes

  Enable input buffer
  Enable normal command processing
  Enable device status reporting

END Accept-Address
```

13.4.2.4.2 Present-Device-Identifier Procedure

Present-Device-Identifier presents the four fields of the **device's** identifier to the **controller.** If all four fields match, Present-Device-Identifier accepts the **device's** new address from the **controller.** If any field does not match, Present-Device-Identifier is exited.

```
BEGIN Present-Device-Identifier
  Set current-field to manufacturer
  Set participating TRUE
  WHILE participating is TRUE
     BEGIN
       Perform Present-Identifier-Field
       (* Present-Identifier-Field will set participating FALSE if there is
       another device on the bus whose identifier comes prior
       to this device's identifier in the search order.*)
       IF participating is TRUE
         THEN
           BEGIN
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
            IF current-field is address
              THEN
                BEGIN
                  Set device-detected to TRUE
                  Set participating to FALSE
                END
              ELSE
                CASE current-field OF
                  BEGIN
                    manufacturer: Set current-field to model number
                    model number: Set current-field to serial number
                    serial number: Set current-field to address
                  END
        END
END

IF device-detected is TRUE
  THEN
    BEGIN
      Read data-byte (* Signals end of identifier through SRQ
                        being FALSE *)
      IF data-byte is 127
        THEN
          BEGIN
            Read data-byte
            Assume primary address indicated by data-byte
            IF device implements extended addressing
              THEN
                BEGIN
                  Read data-byte
                    Assume secondary address indicated by data-byte
                END
          END
        ELSE
          Set device-detected to FALSE
          (* Another device has an address field which is a superset
          of this device's address field *)
   END
END Present-Device-Identifier
```

13.4.2.4.3 Present-Identifier-Field Procedure

Present-Identifier-Field presents successive bytes of the current search field to the **controller**.

```
BEGIN Present-Identifier-Field

  Set identifier-byte-pointer to beginning of current-field
  Set end-of-field to FALSE

   WHILE participating is TRUE AND end-of-field is FALSE
     BEGIN
       Perform Present-Identifier-Byte
       (* Present-Identifier-Byte will set participating FALSE if
         there is another device on the bus whose identifier
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
            comes prior to this device's identifier in the
            search order. *)
          IF identifier-byte-pointer is at end of current-field
            THEN
              Set end-of-field to TRUE
            ELSE
Increment identifier-byte-pointer
        END

    IF participating is TRUE
      THEN
        BEGIN
          (* Signals end of this field by returning SRQ FALSE for the
              search of the next character. *)
          Read data-byte
          Set rsv message FALSE
        END

END Present-Identifier-Field
```

13.4.2.4.4 Present-Identifier-Byte Procedure

Present-Identifier-Byte presents one byte of the **device's** identifier to the **controller**. It uses the SR interface function to guide the **controller** through a binary search for the lowest-value identifier byte in the **system**. The **controller** signals the end of the search by presenting the identifier byte with the most significant bit set. Present-Identifier-Byte checks that this byte matches the **device's** identifier byte. The participating flag is set FALSE if the byte does not match. This mismatch will cause a return to Accept-Address, which will discard data bytes until the next pass.

```
BEGIN Present-Identifier-Byte

    Read data-byte
    IF data-byte is 127
      THEN
        BEGIN
          Set rsv message TRUE
          Set DIO8-detect FALSE
          WHILE DIO8-detect is FALSE
            BEGIN
              Read data-byte
              IF data-byte is between 0 and 127
                THEN
                  BEGIN
                    (* The controller is guided through the binary search
                    by the device with the lowest-value byte at
                    the current position within the identifier. *)
                  IF data-byte is GREATER THAN current identifier byte
                    THEN
                      Set rsv message TRUE
                    ELSE
                      Set rsv message FALSE
            END
          ELSE
            BEGIN
              (* The controller has set the most significant bit of
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
                the byte to indicate that it is done searching and
                this is the value of the lowest valued byte in this
                position. It is placing that byte on the bus so
                that the device can determine if it should continue
                participating in the search. *)
                Set rsv message FALSE
                Set DIO8-detect TRUE
                IF data-byte is NOT EQUAL to
                  (current identifier byte +128)
                    THEN
                      Set participating FALSE
            END
        END
    END
ELSE (* A superset field, excluding address, of another device is
          detected *)
  BEGIN
    Set rsv message FALSE
    Set participating FALSE
  END

END Present-Identifier-Byte
```

13.5 Additional Automatic Configuration Techniques

Addresses and identifications of test **system devices** represent only a portion of the information the **controller** needs in order to perform tasks requested by the user. That is, calibration dates, programming codes, etc., are required by the **controller** to ensure that the test requirements of the user are transformed into valid actions by the test **system**.

The basic techniques provided in this section may be utilized by the application programmer to ease the task of configuring and reconfiguring automatic test **systems**. This section describes only a few of the configuration techniques that are possible. For example, the **system** configuration table can be expanded to include service request masks that are automatically sent to the **devices** when the table is created or updated.

13.6 Examples

See Appendix C for application examples.

14. Controller Compliance Criteria

A **device** must have certain capabilities as described in previous sections. This section lists the capabilities that this standard requires in a **system controller**. A **controller** may optionally contain additional capabilities. Any optional **controller** capabilities that are described by this standard are also listed.

Compliance for a **controller** is divided into several areas that are considered separately. A **controller** must satisfy all the required functionality in each of the areas in order to comply with this standard.

14.1 IEEE 488.1 Requirements

A **controller** shall contain these IEEE 488.1 subsets and no others, see Table 14-1.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 14-1—IEEE 488.1 Requirements

IEEE 488.1 Interface Function	IEEE 488.1 Subsets	IEEE 488.2 Section
Source Handshake	SH1	15.1.2
Acceptor	AH1	15.1.3
Talker	if C5, C7, C9, Cll, C17, C19, C21, or C23	15.1.2
	then TS, T6, TE5, or TE6	
	else T7, T8, TE7, or TE8	
Listener	L3, IA, LE3, or LE4	15.1.3
Service Request	if C5, C7, C9, Cll, C17, C19, C21, or C23	
	then SR1	
	else SR0	
Remote Local	RL0 or RL1	
Parallel Poll	PPO or PP1	
Device Clear	DCO or CD1	
Device Trigger	DT0 or DT1	
Controller		15.1.1
System Controller	C1	
Send IFC and Take Charge	C2	
Send REN	C3	
Respond to SRQ	C4	
Send Interface Messages and Take Control Synchronously	one of C5, C7, C9, C11, C13, C15, C17, C19, C21, C23, C25, or C27	

A **controller** shall meet the requirements of IEEE Std 488.1-1987 [4]. It shall also meet all requirements stated in Section 15. of this standard.

14.2 Message Exchange Requirements

14.2.1 Required Control Sequences

Each of the control sequences listed in Table 14-2 and described in Section 16. shall be implemented individually in a **controller**.

14.2.2 Optional Control Sequences

Each of the control sequences listed in Table 14-3 and described in Section 16. may be implemented individually in a **controller**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.3 Protocols

14.3.1 Required Protocols

Each of the **controller** protocols listed in Table 14-4 and described in Section 17. shall be implemented individually in a **controller**.

Table 14-2—Required Control Sequences

Control Sequence	Section	Requires IEEE 488.1 Subset
SEND COMMAND	16.2.1	one of C5-C28
SEND SETUP	16.2.2	one of C5-C28
SEND DATA BYTES	16.2.3	one of C5-C28
SEND	16.2.4	one of C5-C28
RECEIVE SETUP	16.2.5	one of C5-C28
RECEIVE RESPONSE MESSAGE	16.2.6	one of C5-C28
RECEIVE	16.2.7	one of C5-C28
SEND IFC	16.2.8	C2
DEVICE CLEAR	16.2.9	one of C5-C28
ENABLE LOCAL CONTROLS	16.2.10	C3
ENABLE REMOTE	16.2.11	C3
SET RWLS	16.2.12	C3 and one of C5-C28
SEND LLO	16.2.13	C3 and one of C5-C28
READ STATUS BYTE	16.2.18	one of C5-C28
TRIGGER	16.2.19	one of C5-C28

Table 14-3—Optional Control Sequences

Control Sequence	Section	Requires IEEE 488.1 Subject
PASS CONTROL	16.14	C5, C7, C9, or Cll
PERFORM PARALLEL POLL	16.15	C5, C9, C13, C17, C21, or C25
PARALLEL POLL CONFIGURE	16.16	C5, C9, C13, C17, C21, or C25
PARALLEL POLL UNCONFIGURE	16.17	C5, C9, C13, C17, C21, or C25

Table 14-4—Required Protocols

Protocol Keyword	Section
RESET	17.1
ALLSPOLL	17.3

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,`,,,-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.2 Optional Protocols

The controller protocols listed in Table 14-5 and described in Section 17. may be implemented in a controller.

Table 14-5—Optional Protocols

Protocol Keyword	Section
FINDRQS	17.2
PASSCTL	17.4
REQUESTCTL	17.5
FINDLSTN	17.6
SETADD	17.7
TESTSYS	17.8

14.4 Functional Element Handling

A **controller** shall have the capability of sending all the IEEE 488.2 functional syntactic elements described in Section 7. A **controller** shall have the capability to receive all the IEEE 488.2 functional syntactic elements described in Section 8.

14.5 Controller Specification Requirements

Information about how the manufacturer implemented controller requirements relating to this standard shall be supplied with the controller documentation by the manufacturer. This information shall include

1) A list of IEEE 488.1 Interface Function subsets implemented, see Section 15..
2) A list of control sequences that can be performed by the **controller**, see 16.2.
3) A list of protocols that can be performed by the controller, see Section 17.
4) A description of how the various functional elements are sent and received by the **controller**, see 14.4.
5) Any buffer size limitations shall be described, see 7.7.6 and 8.7.9.
6) A description of how timeouts work, see 15.2 and 15.3.2, including
   a) How timeouts are turned on and off
   b) If the timing is done on a byte or message basis
   c) The allowable time values for the timeout

15. IEEE 488.2 Controller Requirements

15.1 Controller Interface Function Requirements

This section describes the IEEE 488.1 interface function requirements of a **controller.** It also specifies the additional requirements of a **controller** that are directly associated with the ten IEEE 488.1 interface functions. These requirements are intended to supplement the IEEE 488.1 specification for the instrument **system** environment as described in this standard.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

15.1.1 Controller Requirements

The **controller** shall have the following Controller function capabilities: **system controller** (C1), send IFC and take charge (C2), send REN (C3), respond to SRQ (C4), send interface messages (one of C5-C27), and take control synchronously (one of C5, C7, C9, C11, C13, C15, C17, C19, C21, C23, C25, C27). Receive control (one of C5-C15), pass control (one of C5-Cll or C17-C23), pass control to self (one of C5, C7, C17, C19), and parallel poll (one of C5, C9, C13, C17, C21, C25) are optional capabilities.

15.1.2 Talker Requirements

For IEEE 488.2 device-specific message transmission, the **controller** shall contain one of the Talker subsets T5-T8 or TE5-TE8. These subsets mandate the basic talker with unaddress if MLA. Optional capability includes responding to a serial poll and having a talk-only mode. Complete Source Handshake (SH1) capability is also required to support the Talker subset.

15.1.3 Listener Requirements

For IEEE 488.2 device-specific message reception, the **controller** shall contain one of the Listener subsets L3-L4 or LE3-LE4. These subsets mandate the basic listener with unaddress if MTA with optional capability to contain a listen-only mode. Complete Acceptor Handshake (AH1) capability is also required to support the Listener subset.

15.1.4 Passing Control Requirements

If a controller is expected to execute the protocol described in 17.4 and 17.5 for passing and returning control, it shall have the the subset capability to “Pass Control” and “Take Control Synchronously.” That is, subsets C1, C2, C3, and C4 are required along with a choice of either C5, C7, C9, or Cll. A **controller** that can pass control shall also contain the T5, T6, TE5, or TE6 Talker function subsets that support serial poll.

15.1.5 Electrical Requirements

A controller shall implement the E2 electrical interface option (see IEEE Std 488.1-1987 [4], Appendix C2). Open collector drivers shall be used to drive the SRQ, NRFD, and NDAC signal lines. Three-state drivers shall be used to drive the DAV, EOI, ATN, REN, and IFC signal lines. When an IEEE 488.1 **device** in an IEEE 488.1 **system** is not in PPAS, the **controller** shall use three-state drivers to drive the DIO1-8 signal lines. If an IEEE 488.1 **device** in the IEEE 488.1 **system** is in PPAS, the **controller** shall use open collector drivers or the high-impedance state of three-state drivers to drive the DI01-8 signal lines as appropriate to the **controller's** current operation.

15.2 Additional IEEE 488.2 Controller Requirements

A **controller** shall have

1) A means of providing low-level bus control via application software to independently allow the user to
   a) Pulse IFC TRUE for greater than 100 μs.
   b) Set the REN signal line either TRUE or FALSE.
   c) Send singly or in any combination any interface message defined in IEEE 488.1.
   d) Send and detect the IEEE 488.1 END message (that is, the EOI signal line TRUE and the ATN signal line FALSE handshook with a data byte).
2) The facility (either directly or via the application program) to input and output the codes, formats, protocols, and common commands as defined in this standard.
3) The facility to sense the state (TRUE or FALSE) of the SRQ line.
4) The facility to sense FALSE to TRUE SRQ line transitions. Timing of application program actions following the transition is beyond the scope of this standard.
5) The facility (either directly or via the application program) to examine the status byte on a bit basis.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6) The facility to detect the error condition of the **controller** attempting to source handshake a byte while all other **devices** are in AIDS and relay the error to the application program.
7) The facility to timeout on controller-to-device and device-to-controller message exchange and have the ability to relay the occurrence of a timeout to the application program. A timeout occurs when a single handshake or a sequence of handshakes are not completed within an allocated time. The application program shall be able to enable and disable this facility. The **controller** documentation shall include information on how timeouts are controlled.

15.3 IEEE 488.2 Controller Recommendations

15.3.1 Monitoring Bus Lines

For diagnostic use, **controllers** may monitor the IEEE 488.1 signal lines and report their status or transitions to the application program. If the controller does not implement the SETADD and FINDLSTN common **controller** protocols, the application programmer needs to monitor NRFD and NDAC to perform these protocols at the application layer.

15.3.2 Timeouts

The capability to vary the value of the timeout is recommended. The range and resolution is **controller** dependent, but shall be documented.

15.3.3 SRQ Interrupts

It is required that a **controller** be able to sense and report FALSE to TRUE transitions of SRQ, see 15.2. In addition, a controller may optionally provide a means for an application program to branch to a specified section of the application program when SRQ changes from FALSE to TRUE. The SRQ servicing routine may delay execution until the current message is completely transmitted or received, it may start executing immediately after the current data byte is completely transferred, or it may delay execution until a convenient time for the **controller**.

16. Controller Message Exchange Protocols

Fig 16-1 contains the logical model of a **controller**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

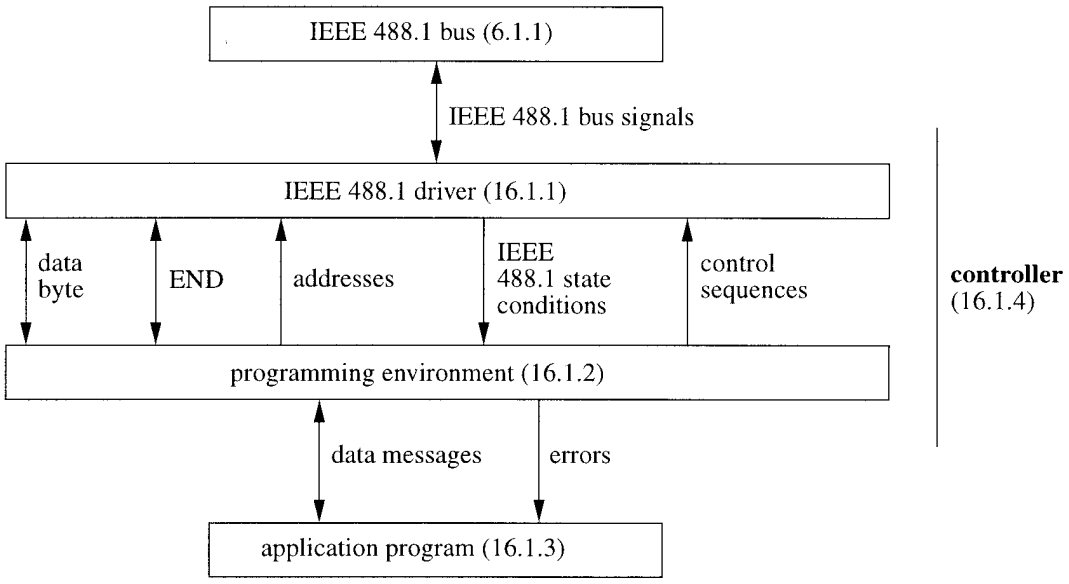


Figure 16-1—Controller Model

16.1 Definitions

16.1.1 IEEE 488.1 Driver

This element contains the state machines described in IEEE Std 488.1-1987 [4]. It also controls much of the sequencing of remote messages.

16.1.2 Programming Environment

This block provides capabilities related to sending and receiving messages on the IEEE 488.1 bus to the application program. This standard describes which capabilities shall be supplied to the application program. The programming environment is typically not written by the user or developer of the **system**. It is, instead, written by the manufacturer or supplier of the **controller**. An operating **system** and programming language with possibly some extra routines would typically comprise a programming environment.

16.1.3 Application Program

The application program is typically written by the user or developer of the **system**. This program is written to perform a specific task in a limited environment. The purpose or description of any application programs is beyond the scope of this standard.

16.1.4 IEEE 488.2 Controller

The controller, described in Section 15., is the combination of the programming environment and the IEEE 488.1 driver. The application program interfaces with the **controller** to perform IEEE 488.1 operations. Many aspects of this combination are beyond the scope of this standard. Only those aspects that are needed to ensure reliable communication between the application program and the **devices** are specified.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16.1.5 IEEE 488.1 Bus Signals

The IEEE 488.1 standard describes messages that are transmitted on the bus and exchanged between the **device** and the bus. This element encompasses those messages.

16.1.6 DAB

See 6.1.4.2.1.

16.1.7 END

See 6.1.4.2.2.

16.1.8 Control Sequence

Control sequences are requests to send IEEE 488.1 remote messages. Control sequences put the state machines in both the **controller** and **devices** into certain desired states.

16.1.9 Addresses

Addresses refer to talk and listen addresses of **devices** and **controllers** on the bus. Addresses are always parameters to a control sequence.

An address has two components. The first component is the primary address and the second component is the secondary address. If the second component is null (a zero secondary address is not null) only the primary address is used.

A **controller** shall provide the user with the capability to send only primary addresses as well as the capability to send both primary and secondary addresses (extended addressing).

This standard places no requirements on how addresses are passed from the application program to the **controller**. The **controller** may require that the application program supply the address explicitly with every request to transfer a message. Alternatively, a data structure may exist that allows a very sophisticated method of determining an address. Many techniques exist, and may be used, to generate addresses within the **controller**.

In general, passing the entire seven bits of a complete primary or secondary address is not necessary. A number between zero and thirty could be passed along with the information that it is either a talk, listen, or secondary address.

16.1.10 IEEE 488.1 State Conditions

The IEEE 488.1 driver contains state machines described in IEEE 488.1. State conditions are indications of which states the state machines are in at any given time. The operating system or programming language occasionally needs to interrogate the current state conditions to be sure messages are being reliably transferred.

16.1.11 Data Messages

Data messages are either <PROGRAM MESSAGE> or <RESPONSE MESSAGE> functional elements.

16.1.11.1 <PROGRAM MESSAGE>

A <PROGRAM MESSAGE> is a sequence of data bytes that is transferred from the **controller** to a **device** with ATN FALSE. The meaning of a <PROGRAM MESSAGE> is, in general, device-specific, see Section 7.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

16.1.11.2 <RESPONSE MESSAGE>

A <RESPONSE MESSAGE> is a sequence of data bytes that is transferred from the **device** to the **controller** with ATN FALSE. The meaning of a <RESPONSE MESSAGE> is, in general, device-specific, see Section 8.

16.1.12 Controller Errors

Controller errors are indications to the application program that the controller was unable to completely perform a request from the application program.

16.2 Control Sequences

Control sequences send one or more IEEE 488.1 remote messages. This standard describes those sequences that are known to be valuable to application programmers. Additional sequences may also be made available to the application program that are beyond the scope of this standard.

Some control sequences are combinations of other control sequences. If a **controller** supplies a control sequence that is the combination of several other control sequences, the **controller** shall also supply the subordinate control sequences individually.

16.2.1 SEND COMMAND

The SEND COMMAND control sequence allows the application program to send an arbitrary sequence of ATN-true commands. This control sequence allows the user to construct control sequences that are not described in this standard or that may not be supplied with the **controller**. One or more 7 bit commands shall be passed with the SEND COMMAND control sequence. The IEEE 488.1 driver shall

```
set ATN TRUE and EOI FALSE
FOR each command passed
  send the command with DIO8 FALSE
NEXT command
```

ATN shall be held TRUE and EOI held FALSE until another control sequence makes them change.

16.2.2 SEND SETUP

The SEND SETUP control sequence configures the **system bus** so that a <PROGRAM MESSAGE> can be transferred from the **controller** to a **device**. It is also used in some other control sequences.

One or more listen addresses shall be passed with the SEND SETUP control sequence. The IEEE 488.1 driver shall

```
set ATN TRUE and EOI FALSE
send the controller's talk address
send the IEEE 488.1 unlisten message (UNL)
FOR each listen address passed
  send the listen address
NEXT listen address
```

ATN shall be held TRUE and EOI held FALSE until another control sequence makes them change.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16.2.3 SEND DATA BYTES

The SEND DATA BYTES control sequence is used to transfer DABs from a **controller** to a **device**. No addresses are passed with the request to perform a SEND DATA control sequence The application program shall, however, pass a list of zero or more DABs and a terminator. The IEEE 488.1 driver shall

set ATN FALSE
FOR each data byte passed
  send a data byte
NEXT data byte
send the terminator

NOTE — If the terminator is the END message, the terminator is actually sent with the last data byte and not separately. If the terminator is the END message and no bytes are passed, the controller shall indicate an error to the application program.

ATN shall remain FALSE while the data bytes and terminator are transferred.

At the completion of the sequence, ATN is FALSE and all currently addressed **devices** remain addressed.

IEEE 488.2 <PROGRAM MESSAGE> elements, see Section 7., are commonly terminated by NL with END. A **controller** shall provide a means for the application program to send an IEEE 488.1 device-specific message without having first specified a message terminator. If the application program has not specified a terminator, the **controller** shall send NL with END as the terminator. A **controller** is also required to have the capability to send END with the last DAB as a terminator. The **controller** may also allow the application program to select alternative terminators.

At least four useful terminators exist:

1) NL with END — required.
2) Send the last DAB with END — required. This terminator allows messages to be sent to IEEE 488.1 **devices** that; terminate only on the END message
3) Null, that is, send nothing — required. The null terminator allows a single message to be sent in pieces via multiple SEND DATA BYTES.
4) Send one or more application program supplied special characters (for example, NL or CR LF) without END — optional. This terminator allows messages to be sent to non-IEEE 488.2 **devices** that relay attach a special significance to the END message.

When sending data messages to **devices**, the application program should use the first alternative.

16.2.4 SEND

The SEND control sequence is used to transfer a complete <PROGRAM MESSAGE> from a **controller** to a **device.**

One or more listen addresses shall be passed with the SEND control sequence. A list of one or more data bytes and a terminator shall also be passed. The method of passing the required information is purposely left unspecified. The terminator shall default to NL with END. The IEEE 488.1 driver shall

execute a SEND SETUP
  with the supplied listen addresses
execute a SEND DATA BYTES
  with the supplied data bytes and terminator

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

16.2.5 RECEIVE SETUP

The RECEIVE SETUP control sequence configures the system bus so that a <RESPONSE MESSAGE> can be transferred from a **device** to the **controller**. It is also used in some other control sequences.

A single talk address shall be passed with the RECEIVE SETUP control sequence. The IEEE 488.1 driver shall

set ATN TRUE and EOI FALSE
send the IEEE 488.1 unlisten message (UNL)
send the **controller's** listen address
send the supplied talk address

ATN shall be held TRUE and EOI held FALSE until another control sequence makes them change.

16.2.6 RECEIVE RESPONSE MESSAGE

The RECEIVE RESPONSE MESSAGE control sequence is used to transfer data bytes from a **device** to a **controller**.

No addresses are passed with the request to perform a RECEIVE RESPONSE MESSAGE control sequence. The **controller** shall be passed a stop-handshaking condition either as a default condition or explicitly. The programming environment shall provide a means of transferring the received information to, the application program. The **controller** may transform or separate the data bytes into a form more readily usable by the application program. For example, a sequence of digits could be converted into the internal representation of a number.

The **controller** shall continue to handshake data bytes until the specified stop-handshaking condition occurs. The controller shall not handshake any data bytes after the stop-handshaking condition until the application program initiates another request to receive data. The IEEE 488.1 driver shall

set ATN FALSE
UNTIL stop-handshaking condition
  receive data bytes
END UNTIL

ATN shall remain FALSE while the data bytes are transferred.

At the completion of the sequence, ATN is FALSE and all currently addressed **devices** remain addressed. The **controller** shall keep NRFD and NDAC TRUE until another control sequence is executed.

A typical stop-handshaking condition for RECEIVE RESPONSE MESSAGE is receiving the END message. A **controller** shall provide the capability to stop-handshaking on the END message.

Alternate stop-handshaking conditions shall be provided that can be selected by the application program. Several ways of stopping the reception of data exist:

1) Stop only on END — required. Works with all IEEE 488.2 **devices** and many non-IEEE 488.2 **devices**.
2) Stop on NL — required. Allows the controller to terminate on the NL at the end of a <RESPONSE MESSAGE>. Also included for compatibility with **devices** that do not adhere to this standard.
3) Stop on comma — required. Allows the **controller** to receive <RESPONSE DATA> elements on an individual basis.
4) Step on semicolon — required. Allows the **controller** to receive <RESPONSE MESSAGE UNIT> elements on an individual basis.
5) Stop on a byte count — optional. May be useful in receiving <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6) Stop on an arbitrary DAB match — optional. Included to allow **controllers** to stop-handshaking at the end of responses from **devices** that do not follow IEEE 488.2 and, thus, may use unknown methods to terminate a response message.
7) Stop at end of <RESPONSE HEADER> or <RESPONSE DATA> — optional. For example, any of the following:
   a) <RESPONSE HEADER SEPARATOR>
   b) <RESPONSE DATA SEPARATOR>
   c) <RESPONSE MESSAGE UNIT SEPARATOR>
   d) <RESPONSE MESSAGE TERMINATOR>
8) Stop on end of <RESPONSE MESSAGE DATA ELEMENT> — optional. For example, any of the following:
   a) <RESPONSE MESSAGE DATA SEPARATOR>
   b) <RESPONSE MESSAGE UNIT SEPARATOR>
   c) <RESPONSE MESSAGE TERMINATOR>
9) Stop on end of <RESPONSE MESSAGE UNIT> — optional. For example, any of the following:
   a) <RESPONSE MESSAGE UNIT SEPARATOR>
   b) <RESPONSE MESSAGE TERMINATOR>

Implementation of options (7), (8), and (9) is recommended, as they allow an application program to easily identify the syntactic components of a <RESPONSE MESSAGE>.

A **controller** designer may chose to allow various combinations of conditions to generate the stop-handshaking condition. For example, the stop-handshaking condition could be NL or END, NL or semicolon, etc. This list is not intended to restrict the **controller** from supplying additional techniques for stopping the handshaking process.

NOTE — Alternatives (7), (8), and (9) require the **controller** to parse the received message in order to detect the stop-handshaking condition.

16.2.7 RECEIVE

The RECEIVE control sequence is used to transfer a <RESPONSE MESSAGE> from a **device** to a **controller**.

A single talk address and a stop-handshaking condition shall be passed with the RECEIVE control sequence. The application program shall also provide internal storage for the received data bytes. The IEEE 488.1 driver shall

execute a RECEIVE SETUP
with the supplied talk address
execute a RECEIVE RESPONSE MESSAGE
with the supplied stop-handshaking condition

16.2.8 SEND IFC

The SEND IFC control sequence can only be executed by the **system controller**. The effect of the control sequence is to remove all talkers and listeners, serial poll disable all **devices**, and return control to the **system controller**. At power-on, the controller shall execute the SEND IFC control sequence.

When the application program requests that the SEND IFC control sequence be executed, the IEEE 488.1 driver shall pulse the IEEE 488.1 bus signal IFC TRUE for more than 100 s (see IEEE 488.1-1987 [4], 2.12.3.16).

The state of the other IEEE 488.1 bus lines shall not change. The **controller** shall not execute any other control sequences until IFC has been made FALSE by this control sequence.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

16.2.9 DEVICE CLEAR

The DEVICE CLEAR control sequence is used to send the **dcas** message and place all or selected **devices** in Device Clear Active State (DCAS). The behavior of a **device** in DCAS is well defined, see 5.8. The DEVICE CLEAR control sequence should be used as part of the initialization of a **system**, see 17.1.

After completion of the DEVICE CLEAR control sequence, the DIO lines may be left in any state convenient for the **controller**. ATN shall remain TRUE at the end of the control sequence. The EOI line shall remain FALSE during and at the completion of the DEVICE CLEAR control sequence to avoid performing parallel polls.

16.2.9.1 Selected Device(s)

If the application program passes an address(es) with a request to send the DEVICE CLEAR control sequence, the IEEE 488.1 driver shall

execute a SEND SETUP
with the supplied addresses
send the IEEE 488.1 Selected Device Clear message (SDC)

16.2.9.2 All Devices

If the application program requests that the DEVICE CLEAR control sequence be sent, and no addresses are also passed, the IEEE 488.1 driver shall

set ATN TRUE
send the IEEE 488.1 Device Clear message (DCL)

16.2.10 ENABLE LOCAL CONTROLS

The ENABLE LOCAL CONTROLS control sequence is used to place either all or selected **devices** into a local state (LOCS or LWLS). See 5.6 for a discussion on how **devices** behave after a transition from a remote to a local state and while in a local state.

16.2.10.1 Selected Device(s)

If the application program passes an address(es) with a request to execute the ENABLE LOCAL CONTROLS control sequence, the IEEE 488.1 driver shall

execute a SEND SETUP
with the supplied addresses
send the IEEE 488.1 “go to local” message (GTL)

After completion of the ENABLE LOCAL CONTROLS control sequence for selected **devices**, the DIO lines may be left in any state convenient for the **controller**. ATN shall remain TRUE. The EOI line shall remain FALSE during and at the completion of the ENABLE LOCAL CONTROLS control sequence to avoid performing parallel polls.

16.2.10.2 All Devices

If the application program requests that the ENABLE LOCAL CONTROLS control sequence be sent, and no addresses are also passed, the IEEE 488.1 driver shall

set the IEEE 488.1 signal REN FALSE
(enter the C interface function systemcontrol remote
not active state, see IEEE 488.1-1987 [4], 2.12.3.18)

The state of the other IEEE 488.1 bus lines shall not change.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16.2.11 ENABLE REMOTE

The ENABLE REMOTE control sequence allows the application program to put the IEEE, 488.1 **system** or selected **devices** into a remote state. This sequence is a companion to the ENABLE LOCAL CONTROLS control sequence.

16.2.11.1 Selected Device(s)

If the application program passes an address(es) with the request to send the ENABLE REMOTE control sequence, the IEEE 488.1 driver shall

IF REN is FALSE
  THEN wait until REN has been FALSE at least 100 μs
set the IEEE 488.1 signal line REN TRUE
execute a SEND SETUP
        with the supplied addresses

16.2.11.2 All Devices

If the application program does not pass an address with the request to send the REMOTE control sequence, the **controller** shall

IF REN is FALSE
  THEN wait until REN has been FALSE at least 100 s
set the IEEE 488.1 signal line REN TRUE

The state of the other IEEE 488.1 bus lines shall not change. Note that merely making REN TRUE does not make a **device** change states. The **device** must later be addressed to listen before it will enter one of the remote states (REMS or RWLS).

16.2.12 SET RWLS

The SET RWLS control sequence allows the application program to disable the return to local (rtl) local message in one or more **devices**. Selected **devices** are put into RWLS. **Devices** not currently addressed to listen are put into LWLS by this control sequence.

One or more listen addresses are passed with a request to send the SET RWLS control sequence. The IEEE 488.1 driver shall

execute an ENABLE REMOTE
        with the supplied listen addresses
send the IEEE 488.1 local lockout message (LLO)

After completion of the SET RWLS control sequence, the DIO lines may be left in any state convenient for the **controller**. The ATN line shall remain TRUE and the EOI line FALSE until another control sequence changes them.

16.2.13 SEND LLO

The SEND LLO control sequence allows the application program to put all the **devices** on the bus that are in LOCS into LWLS. Any **device** that happened to be in LADS when this control sequence is executed will actually be put into RWLS.

No addresses are passed with the request to perform a SEND LLO control sequence. The IEEE 488.1 driver shall

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

execute an ENABLE, REMOTE without addresses
set ATN TRUE and EOI FALSE
send the IEEE 488.1 local lockout message (LLO)

After completion of the SEND LLO control sequence, the DIO lines may be left in any state convenient for the **controller**. The ATN line shall remain TRUE and the EOI line FALSE until another control sequence changes them.

16.2.14 PASS CONTROL

The PASS CONTROL control sequence allows the application program to give control of the **system bus** to a **device**. A protocol is provided in 17.4 for transferring control.

The application program shall pass a talk address whenever it requests the **controller** to execute the PASS CONTROL control sequence. The IEEE 488.1 driver shall

execute a RECEIVE SETUP
with the supplied talk address
send the IEEE 488.1 take control message (TCT)
set ATN FALSE

NOTE — Control is passed as a consequence of the IEEE 488.1 C function protocol. Handshaking the TCT causes the controller that is passing control to enter CIDS. ATN going false then causes the receiving **controller** to enter CACS, become active, and assert ATN.

16.2.15 PERFORM PARALLEL POLL

The PERFORM PARALLEL POLL control sequence allows the application program to perform a parallel poll on the **system bus**. No addresses are passed with the request to perform a PERFORM PARALLEL POLL control sequence. The application program shall, however, supply a storage location for the resulting parallel poll response (PPR) message. When the application program requests that a PERFORM PARALLEL POLL control sequence be sent, the IEEE 488.1 driver shall

set ATN TRUE and EOI TRUE
read and store the resulting PPR message in
the supplied storage location
send rpp local message FALSE

16.2.16 PARALLEL POLL CONFIGURE

The PARALLEL POLL CONFIGURE control sequence allows the application program to configure the parallel poll response from a **device**. The **device** being configured must have IEEE 488.1 PP1 capability.

The application program shall pass an address, the DIO line to be driven, and the sense of the driven line with the request for a PARALLEL POLL CONFIGURE control sequence. An appropriate parallel poll enable message is constructed from the information about which line is to be driven and the sense of the driven line. The IEEE 488.1 driver shall

execute a SEN SETUP
with the supplied address
send the IEEE 488.1 parallel poll configure message (PPC)
send the IEEE 488.1 parallel poll enable message (PPE)

At the completion of the control sequence, ATN shall remain TRUE and EOI shall remain FALSE until changed by another control sequence.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16.2.17 PARALLEL POLL UNCONFIGURE

The PARALLEL POLL UNCONFIGURE control sequence provides a means to keep all or selected **devices** from responding to a parallel poll. At the completion of the control sequence, ATN shall remain TRUE and EOI shall remain FALSE until changed by another control sequence.

16.2.17.1 Selected Device(s)

If the application program passes an address(es) with the request to send the PARALLEL POLL UNCONFIGURE control sequence, the IEEE 488.1 driver shall

execute a SEND SETUP
    with the supplied addresses
send the IEEE 488.1 parallel poll configure message (PPC)
send the IEEE 488.1 parallel poll disable message (PPD)
    with the lower four bits as zeros

16.2.17.2 All Devices

If the application program does not pass an address with the request to send the PARALLEL POLL UNCONFIGURE control sequence, the IEEE 488.1 driver shall

set ATN TRUE
send the IEEE 488.1 parallel poll unconfigure message (PPU)

16.2.18 READ STATUS BYTE

The READ STATUS BYTE control sequence provides a means of reading the status byte from a specific **device**.

A single talk address shall be passed with the request. The **controller** shall provide a storage location for the status byte. When the application program requests the controller to send the READ STATUS BYTE control sequence, the IEEE 488.1 driver shall

set ATN TRUE and EOI FALSE
send the IEEE 488.1 unlisten message (UNL)
send the controller's listen address
send the IEEE 488.1 serial poll enable message (SPE)
send the talk address
set ATN FALSE handshake a data byte
store the status byte and RQS message
set ATN TRUE
send the IEEE 488.1 serial poll disable message (SPD)
send the IEEE 488.1 untalk message (UNT)

At the completion of the control sequence, ATN shall remain TRUE and EOI shall remain FALSE until changed by another control sequence. At no time during the control sequence shall the **controller** set EOI TRUE. The **controller** shall ignore the state of the EOI line while handshaking the data byte.

16.2.19 TRIGGER

The TRIGGER control sequence provides a way to send the IEEE 488.1 group execute trigger message to specific **devices** or to all **devices** that are currently addressed to listen.

-`,,`,,,-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

At the completion of the control sequence, ATN shall remain TRUE and EOI shall remain FALSE until changed by another control sequence.

16.2.19.1 Selected Device(s)

If the application program passes an address(es) with the request to send the TRIGGER control sequence, the IEEE 488.1 driver shall

execute a SEND SETUP
with the supplied addresses
send the IEEE 488.1 group execute trigger message (GET)

16.2.19.2 All Addressed Devices

If the application program does not pass an address with the request to send the TRIGGER control sequence, the IEEE 488.1 driver shall

set ATN TRUE
send the IEEE 488.1 group execute trigger message (GET)

NOTE — If a TRIGGER control sequence is executed and no **devices** are currently addressed to listen, no **device** will be triggered. All **devices** on the bus will handshake the GET message, but only those that are currently addressed to listen and have DT1 capability will process. the message.

17. Common Controller Protocols

This section describes protocols that are executed by the **controller** in a **system**. These protocols make use of required capability and occasionally optional capability, which is described in this standard. The protocols are generally designed to work in mixed **systems** consisting of both **devices** and non-IEEE 488.2 **devices**. This standard attempts to point out special considerations that may have to be observed when operating in a **system** containing both **devices** and non-IEEE 488.2 **devices**. Special considerations listed in this section should not be thought of as exhaustive.

Many of the protocols require that information be passed from or to the protocol. The mechanism for passing this information is intentionally left unspecified. The particular technique used for passing information is left to the **controller** designer's discretion.

Each protocol described in this section has a keyword associated with it. **Controller** manufacturers and designers are encouraged to use this keyword in the **controller's** documentation. Each protocol has an associated controller algorithm. A **controller** shall implement a functionally equivalent routine for this algorithm when implementing the protocol. Some of the constructs used in describing the algorithm may not be directly available in the **controller**. The **controller** may also have constructs that are superior to the ones used in the description of the algorithm.

Table 17-1 lists all the common controller protocols defined in this standard.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-1—IEEE 488.2 Common Controller Protocols

Keyword	Name	Section	Compliance
RESET	Reset	17.1	Mandatory
FINDRQS	Find Device Requesting Service	17.2	Optional
ALLSPOLL	Serial Poll All Devices	17.3	Mandatory
PASSCTL	Pass Control	17.4	Optional
REQUESTCTL	Request Control	17.5	Optional
FINDLSTN	Find Listeners	17.6	Optional
SETADD	Set Address	17.7	Optional, but requires FINDLSTN
TESTSYS	Self-Test System	17.8	Optional

17.1 Reset Protocol

17.1.1 Keyword

RESET.

17.1.2 Purpose

The reset protocol is used to initialize an entire **system** using three levels.

1) *Bus initialization*. The first level of reset places the **system bus** into an idle condition. All **devices** are unaddressed and put into a serial poll idle state. The **system controller** becomes controller-in-charge.
2) *Message Exchange initialization*. The second level of reset ensures that a **device** can be sent a <PROGRAM MESSAGE>. See 5.8.
3) *Device initialization*. The third level of reset initialize device-specific functions within a **device**. See 10.32

17.1.3 Information Requested by the Protocol

This protocol requires a list of the addresses of all the **devices** in the **system**.

17.1.4 Information Supplied by the Protocol

Nothing is returned by the protocol.

17.1.5 Controller Algorithm

A complete **system** reset is accomplished by doing all three levels of reset, in order, to every **device**.

1) *Bus initialization*. The **controller** shall execute a ENABLE REMOTE control sequence, see 16.2.11.2, with no addresses. The **controller** shall then execute a SEND IFC control sequence, see 16.2.8.
2) *Message Exchange initialization*. The DEVICE CLEAR control sequence is executed without addresses. This control sequence sends a universal Device Clear. See 16.2.9.2.
3) *Device initialization*. The **controller** shall execute a SEND control sequence with the supplied addresses and the <TERMINATED PROGRAM MESSAGE>:

*RST<PMT>

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.1.6 Additional Requirements and Guidelines

Under different circumstances, each level of reset may be done individually or in combination with other levels. If several levels are to be done together, Bus initialization should be performed before Message Exchange initialization, and Message Exchange initialization should be performed before Device initialization. Specification of all possible combinations is beyond the scope of this standard.

NOTE — **Devices** that do not comply with this standard may behave much differently from what is described in this protocol for message exchange and **device** initialization. A bus initialization, IFC, should behave identically on all IEEE 488.1 **devices**. The behavior of non-IEEE 488.2 **devices** should be investigated before doing message exchange initialization with them. A **device** initialization should never be done to a **device** that has not implemented the *RST command.

17.1.7 Standard Compliance

The RESET common **controller** protocol shall be supplied in all **controllers.**

17.2 Find Device Requesting Service Protocol

17.2.1 Keyword

FINDRQS.

17.2.2 Purpose

The purpose of the FINDRQS protocol is to find a **device** that is requesting service and to return its status byte.

17.2.3 Information Requested by the Protocol

A list containing the address of every **device** that can request service must be supplied in the order in which they should be polled. A **device** with IEEE 488.1 SR1 capability must be at every address in the list.

The list may be explicitly passed to the protocol or may reside in a common area. This protocol need only know where to find the address list.

17.2.4 Information Supplied by the Protocol

When the protocol has been completed successfully, the address of the first **device** in the list requesting service and its status byte are returned. If not successful, an error message is returned.

17.2.5 Controller Algorithm

The protocol is initiated when SRQ is sensed TRUE. The application program may test the SRQ line and then invoke the protocol, or the **controller** may automatically initiate the protocol when SRQ is true. The time from SRQ becoming TRUE to the initiation of this protocol is intentionally unspecified.

```
BEGIN Find Device Requesting Service
   Initialize pointer to top of address list
   Assert ATN TRUE
   Send IEEE 488.1 UNL remote message
   Send controller's LAG
   Send IEEE 488.1  SPE remote message
   REPEAT
      Send TAG of address pointed to
      Set ATN FALSE
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
      Handshake a DAB (STB & RQS)
      Advance pointer
      Set ATN TRUE
   UNTIL RQS is true or pointer is past the end of the address list
   Send IEEE 488.1  SPD remote message
   Send IEEE 488.1  UNT remote message
   IF RQS is TRUE
      THEN return last address sent
         return last status byte received
      ELSE return error
END Find Device Requesting Service
```

17.2.6 Additional Requirements and Guidelines

The ordering of the addresses in the list is important. The **device** that requires the most rapid servicing should have its address at the top of the list.

The protocol cannot serial poll all possible IEEE 488.1 addresses for at least two reasons. First, serial polling an address that does not have IEEE 488.1 SR1 capability may cause the handshake never to be completed. Second, the time required to poll all the primary addresses and secondary addresses is prohibitively long even if timeouts are used.

Certain precautions must be, taken to ensure reliable performance of this protocol.

The service request interface function in IEEE 488.1 allows a **device** to change the SRQ remote message sent without receiving any other remote messages. If a **device** changes its rsv message from TRUE to FALSE during the execution of the protocol, an error may be generated. **Devices** are constrained not to remove the local rsv message until specific remote messages are received. If the FINDRQS is initiated by an interrupt, multitasking operating systems need to be told which task is to be passed the information obtained by this protocol.

At the completion of this protocol, all **devices** will be unaddressed. An error will occur in the application program if **devices** are expected to still be addressed when control is returned from the protocol.

This protocol makes no provision for the case when the address of a **device** is included in the list that does not have SR1 capability. If the **device** has T3, T4, T7, or T8 capability, the **device** may source handshake a data byte from its normal output queue. DIO7 in this byte has no relation to RQS. If the **device** has SH0 or TO capability, or has nothing to say when it is addressed to talk, the **device** will never source handshake a data byte, and this protocol will never complete unless a timeout is implemented.

17.2.7 Standard Compliance

The FINDRQS common **controller** protocol may optionally be supplied in **controllers**.

17.3 Serial Poll All Devices Protocol

17.3.1 Keyword

ALLSPOLL.

17.3.2 Purpose

The purpose of the ALLSPOLL protocol is to read the status byte of every **device** with SR1 capability in the **system**.

--`,,,`-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

17.3.3 Information Requested by the Protocol

A list containing the address of every **device** that has SR1 capability should be supplied to the protocol. The ordering of the addresses in the list is not important. A **device** with IEEE 488.1 SR1 capability should be at every address in the list. Storage locations for all the status bytes to be read must also be supplied.

The list may be explicitly passed to the protocol or may reside in a common area. This protocol must only know where to find the list of addresses.

17.3.4 Information Supplied by the Protocol

When the protocol has been completed successfully, the status byte of every **device** is available. The status bytes shall be stored in a way such that the application program can associate a particular status byte with an address.

17.3.5 Controller Algorithm

The protocol may be initiated at any time by the application program. The sense of SRQ when the protocol is invoked is unimportant.

```
BEGIN Serial Poll All Devices
   Initialize pointer to top of address list
   Initialize status byte list
   Set ATN TRUE
   Send IEEE 488.1 UNL remote message
   Send controller's LAG
   Send IEEE 488.1 SPE remote message
   WHILE pointer is not past the end of the address list
      Send TAG of address pointed to
      Set ATN FALSE
      Handshake a DAB (STB & RQS)
      Store the STB & RQS in the status byte list
      Advance pointer
      Set ATN TRUE
   END WHILE
   Send IEEE 488.1 SPD remote message
   Send IEEE 488.1 UNT remote message
END Serial Poll All Devices
```

17.3.6 Additional Requirements and Guidelines

The protocol cannot serial poll all possible IEEE 488.1 addresses for at least two reasons. First, serial polling an address that does not have IEEE 488.1 SR1 capability may cause the handshake never to be completed. Second, the time required to poll all the primary addresses and secondary addresses is prohibitively long even if timeouts are used.

The service request interface function in IEEE 488.1 allows a **device** to send multiple STBs while in SPAS. This protocol will only read the first one.

At the completion of this protocol, all **devices** will be unaddressed. An error will occur in the application program if **devices** are expected to still be addressed when control is returned from the protocol.

This protocol makes no provision for the case when the address of a **device** is included in the list that does not have SR1 capability. If the **device** has T3, T4, T7, or T8 capability, the **device** may source handshake a data byte from its normal output queue. DI07, in this byte, has no relation to RQS. If the **device** has SR0 or T0 capability, or has nothing

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

to say when it is addressed to talk, the **device** will never source handshake a data byte, and this protocol will never complete unless a timeout is implemented.

17.3.7 Standard Compliance

The ALLSPOLL common **controller** protocol shall be supplied in all **controllers**.

17.4 Pass Control Protocol

17.4.1 Keyword

PASSCTL.

17.4.2 Purpose

The purpose of this protocol is to pass control among the **devices** that have **controller** capability and need to be controller-in-charge.

17.4.3 Information Requested by the Protocol

The protocol requires the address of a **device** that is currently requesting control of the bus. The protocol must also have the address of the **controller** executing the protocol.

17.4.4 Information Supplied by the Protocol

The address of the **device** that was passed control is maintained by this protocol.

17.4.5 Controller Algorithm

```
BEGIN Pass Control
  IF using primary addressing only
    THEN
      Execute SEND control sequence with
        *PCB(SP)<NRI> as the <PROGRAM MESSAGE>
        and NL^END as the terminator.
        (* NR1 is the primary address of the
           controller-in-charge that is passing control. *)
    ELSE (* using primary and secondary addressing *)
      Execute SEND control sequence with
        *PCB(SP)<NRI>,<NRI> as the <PROGRAM MESSAGE>
         and NL^END as the terminator.
        (* first NR1 is the primary address and
        the second NR1 is the secondary address of the
        controller-in-charge that is passing control. *)
  Execute PASS CONTROL used as a control sequence with the talk address of
    device requesting control.
END Pass Control
```

When the **device** no longer requires control of the bus, it shall pass control using a PASS CONTROL control sequence with the address it received with the *PCB command. See 10.21 and 16.2.14 for details of the *PCB command.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

17.4.6 Additional Requirements and Guidelines

When sending its address using the *PCB command, a **controller** shall send integers in the range of 0 to 30 in the primary address field and integers in the range of 0 to 30 in the secondary address field. If no secondary addressing is to be used when passing control back, the second parameter shall be omitted.

The controller-in-charge is responsible for determining when a **device** is requesting control of the **system bus**. The request control bit in the Standard Event Status Register may be enabled to cause an SRQ, so that the **controller** may use SRQ to look for a **device** requesting control. Alternatively, the **controller** may periodically read the Standard Event Status Register to determine if a **device** is requesting control.

The **system controller** may use a SEND IFC control sequence to regain control of the **system bus**. This method of gaining control should only be used when an error condition is known to exist.

The current controller-in-charge, even if it is not the **controller**, may send the *PCB command to another potential controller-in-charge. The current controller-in-charge may then pass control in a linked list structure.

Many possibilities for passing control among several **controllers** on the same bus exist. An attempt to identify all the possible techniques of passing control in such a **system** is beyond the scope of this standard.

17.4.7 Standard Compliance

The PASSCTL common **controller** protocol may optionally be supplied in **controllers**.

17.5 Requesting Control

17.5.1 Keyword

REQUESTCTL.

17.5.2 Purpose

The purpose of this protocol is to allow a potential controller-in-charge to indicate the need for control to the current controller-in-charge; to receive the control; and then to relinquish that control when it no longer requires control of the **system bus**. The protocol may be included in **devices** that have **controller** capability.

17.5.3 Information Requested by the Protocol

None.

17.5.4 Information Supplied by the Protocol

None.

17.5.5 Controller Algorithm

The component (**device** or **controller**) requesting to be controller-in-charge asserts its Request Control bit TRUE in its Standard Event Status Register to invoke the protocol.

The current controller-in-charge detects the request in the components Standard Event Status Register, either as a result, of an SRQ indication by the component or by a polling routine, and executes the PASSCTL **controller** protocol sequence. In this sequence, it sends its address (or the address of another component) to which the requesting component is to later pass control. It then passes control to the requesting component.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The requesting component stores the address it received to which it is to pass control and then accepts control. It sends/receives messages on the bus until it is ready to relinquish control. The component then executes a PASSCTL control sequence to relinquish control to the component designated by the stored address.

The component that is to receive control participates in the PASSCTL control sequence and gains control of the bus.

17.5.6 Additional Requirements and Guidelines

A **device** that does not have **controller** capability shall not request control. A **device** with **controller** capability is required, see 4.8, to have the capability to pass control.

The current controller-in-charge, even if it is not the **controller**, may send the *PCB command to another potential controller-in-charge. The current controller-in-charge may then pass control in a linked list structure.

Many possibilities for passing control among several **controllers** on the same bus exist. Attempting to identify all the possible techniques of passing control in such a system is beyond the scope of this standard

17.5.7 Standard Compliances

The REQUESTCTL common **controller** protocol may optionally be supplied in **controllers**.

17.6 Find Listeners Protocol

17.6.1 Keyword

FINDLSTN.

17.6.2 Purpose

The Find Listeners protocol identifies each IEEE 488.1 address at which at least one listener **device** resides.

17.6.3 Information Supplied to the Protocol

This protocol requires a list of primary addresses that are to be searched.

17.6.4 Information Supplied by the Protocol

This protocol returns a list of primary addresses, and, if extended addresses exist, primary/secondary address pairs at which **devices** reside.

17.6.5 Controller Algorithm

```
BEGIN Find Listeners
  WHILE a primary address remains unsent in the address input list
    BEGIN
      Send the IEEE 488.1  Unlisten message (UNL)
      Send the LAG of the primary address
      Set ATN FALSE
      Wait 1.5 ms minimum
      IF NDAC is TRUE
        THEN
          Place primary address in output address list
        ELSE
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
        BEGIN
          Send all secondary addresses
          Set ATN FALSE
          Wait 1.5 ms minimum
          IF NDAC is TRUE
            THEN
              BEGIN
                Send the IEEE 488.1 Unlisten message (UNL)
                Send the LAG of the primary address
                WHILE a secondary address has not been tested
                  BEGIN
                    Send an untested secondary address
                    Set ATN FALSE
                    Wait 1.5 ms minimum
                    IF NDAC is TRUE
                      THEN
                        BEGIN
                          Put extended address in output list
                          Send the IEEE 488.1 Unlisten message (UNL)
                          Send the LAG of the primary address
                        END
                  END
              END
        END
    END
  Send the IEEE 488.1  Unlisten message (UNL)
END Find Listeners
```

17.6.6 Additional Requirements and Guidelines

The 1.5 ms wait time may not be sufficient for some **devices** because the time to enter Acceptor Idle State (AIDS) of the Acceptor Handshake Function is not specified in IEEE 488.1 .

This protocol will not work properly in **systems** containing talk and/or listen-only **devices**. For example, some bus analyzers and bus extenders appear as listen-only **devices**, which will cause the protocol to indicate that every address is occupied.

17.6.7 Standard Compliance

The FINDLSTN common **controller** protocol may be optionally supplied in a **controller**.

17.7 Set Address Protocol

17.7.1 Keyword

SETADD.

17.7.2 Purpose

The purpose of this protocol is to detect address-configurable **devices** (those implementing the *AAD common command) and to assign these **devices** primary listen or extended listen addresses in accordance with the information given by a user-supplied configuration table.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.7.3 Information Requested by the Protocol

This protocol requires a list of addresses to be searched and a configuration table. The table shall contain a set of entries in which each entry has a set of fields consisting of a manufacturer, model number, serial number, and the listen address to be assigned. An entry is allowed to have empty fields.

17.7.4 Information Supplied by the Protocol

This protocol supplies an updated configuration table of those **devices** detected in the address space given in the input address list and error status.

17.7.5 Controller Algorithm

NOTES:

1 — Four fields are searched as shown below. The first the fields are identical to the associated fields of the *IDN?. response. The fourth is determined by the **devices** current address.

Manufacturer
Model Number
Serial Number
Current Primary or Extended Listen Address

2 — All data bytes are sent to the **system bus** with DIO8 FALSE, unless otherwise specified.

3 — The primary listen address assigned by this protocol is sent as a single data byte having a decimal value in the range of 32 through 62. The secondary address is sent as a data byte in the range of 96 through 126. A primary or extended address is referred to as a listen address in this protocol.

17.7.5.1 Set Address Procedure

```
BEGIN Set-Address
  Mark all entries of configuration table as “unmatched”
  Mark each entry that has at least one nonempty field as “nonempty”
  IF any addresses in address list
    THEN
      Perform SEND SETUP using all addresses in input list
      Send SDC
      Perform SEND DATA BYTES using the *AAD <PROGRAM MESSAGE> and
        the NL^END <PROGRAM MESSAGE TERMINATOR>
      Wait 100 ms minimum
      Set device-search to TRUE
      WHILE device-search is TRUE
        BEGIN
          Set current-field to manufacturer
          Set data-byte to ASCII STX (* decimal value of 2 *)
          Perform SEND DATA BYTES using data-byte
          Perform Acquire-Field
          IF field-detected is TRUE
            THEN
              BEGIN
                WHILE field-detected is TRUE
                  BEGIN
                    CASE current-field OF
                      BEGIN
                        manufacturer: set current-field to model number
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
                        model number: set current-field to serial number
                        serial number: set current-field to address
                        address: set current-field to done
                        OTHERWISE: set current-field to extra field
                      END
                    Perform Acquire-Field
                  END
                IF current-field equals done
                  THEN
                    Perform Configure-Device
                  ELSE
                    BEGIN
                      Generate error status indicating that the detected
                        device has an improperly-formed identifier
                      Perform Error-Exit
                    END
              END
            ELSE
              Set device-search to FALSE
      END
    Send SDC
END Set-Address
```

17.7.5.2 Error Exit Procedure

```
BEGIN Error-Exit
  Send SDC
  Return error status and control to calling program
END Error-Exit
```

17.7.5.3 Acquire Field Procedure

```
BEGIN Acquire-Field
  Set field-data-byte-pointer to beginning of current-field
  Perform Acquire-Byte
  IF byte-detected is TRUE
    THEN
      BEGIN
      Set field-detected to TRUE
        WHILE byte-detected is TRUE
          Perform Acquire-Byte
      END
    ELSE
      Set field-detected to FALSE
END Acquire-Field
```

17.7.5.4 Acquire Byte Procedure

```
BEGIN Acquire-Byte
  Set data-byte to decimal 127
  Perform SEND DATA BYTES using data-byte
  Wait 1.5 ms minimum, or until NRFD is FALSE
  IF SRQ is TRUE
    THEN
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
        BEGIN
          Set byte-detected to TRUE
          Set initial data-byte to decimal 64
          Set delta to decimal 32
          WHILE delta is 1 or greater
            BEGIN
              Perform SEND DATA BYTES using data-byte
              Wait 1 ms minimum, or until NRFD is FALSE
              IF SRQ is TRUE
                THEN
                  data-byte = data-byte - delta
                ELSE
                  data-byte = data-byte + delta
              IF data-byte is GREATER THAN decimal 126 or
                data-byte is LESS THAN decimal 32
                  THEN
                    BEGIN
                      Generate error status indicating that an out-of-range
                        byte has been found or some other device has
                        interfered with the protocol by asserting SRQ.
                      Perform Error-Exit
                    END
              delta = delta/2
            END
          Perform SEND DATA BYTES using data-byte
          Wait 1.5 ms minimum, or until NRFD is FALSE
          IF SRQ is TRUE
            THEN
              Decrease value of data-byte by one
          Perform SEND DATA BYTES using data-byte + decimal 128
          Store data-byte in field indicated by current-field at
            position indicated by field-data-byte-pointer
          Increment field-data-byte-pointer
        END
      ELSE
        Set byte-detected to FALSE
END Acquire-Byte
```

17.7.5.5 Configure-Device Procedure

```
BEGIN Configure-Device
  Mark all unmatched nonempty entries as untested
  Set best-matched-entry to no-entry
  Set equal-score to zero
  FOR each nonempty AND untested entry
    BEGIN
      Set equal-fields to zero
      Set entry-match to TRUE
      Set test-field to TRUE
      WHILE test-field is TRUE
        BEGIN
          IF a nonempty entry field has not been tested,
            excluding the address field, in table order
              THEN
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
                    IF the entry field is equal to OR is an initial
                      substring of the corresponding acquired field,
                      ignoring upper/lowercase distinction and leading
                      and trailing white space in either string
                        THEN
                          Increment equal-fields by one
                        ELSE
                          BEGIN
                            Set entry-match to FALSE
                            Set test-field to FALSE
                          END
                    ELSE
                      Set test-field to FALSE
                END
            IF entry-match is TRUE
              THEN
                IF equal-fields is GREATER THAN equal-score
                  THEN
                    BEGIN
                      Set equal-score to equal-fields
                      Set best-matched-entry to point to this entry
                    END
                Mark entry as tested
            END
IF best-matched-entry is no-entry
  THEN
    BEGIN
      Find first entry in table with manufacturer, model number,
        and serial number fields all empty
      IF empty entry does not exist
        THEN
          BEGIN
            Generate error status indicating configuration table
              overflow
            Perform Error-Exit
          END
        ELSE
          BEGIN
            selected-entry is empty entry
            Mark this entry as added and matched
          END
       END
 ELSE
   BEGIN
     selected-entry is best-matched-entry
      Mark selected entry as matched
    END
Fill in manufacturer, model number, and serial number fields
  of selected entry with the corresponding acquired fields
IF the address field of the selected entry is empty
  THEN
    BEGIN
      Store acquired listen address in selected entry address field
      Perform SEND DATA BYTES using acquired listen address
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
            as a data byte(s)
        END
    ELSE
      IF entry address field length does NOT EQUAL acquired
      address length.
        THEN
          Generate error indicating that the table address
            is incompatible with the device address.
          Perform Error-Exit
        ELSE
          Perform SEND DATA BYTES using table entry address as
            a data byte(s)
END Configure-Device
```

17.7.6 Additional Requirements and Guidelines

The input configuration table supplies to the protocol controls the way in which addresses are assigned. If the user supplies an empty table, the protocol returns a table that duplicates the IEEE 488.1 address configuration (within the address space specified in the input address list) prior to the execution of the protocol. That is, after the protocol has finished, the address configuration is unchanged. If all the table entry address fields are filled with addresses, exactly those addresses will be assigned. The addresses may be in any order. If more **devices** are detected than the input table can accommodate, then the **controller** shall indicate an error and leave the addresses of any additional **devices** unchanged. The protocol becomes more selective in assigning addresses as more fields are specified. For example, a manufacturer, model number, and address in a table entry will cause the protocol to assign that address to a **device** that has that manufacturer and model number. An entry with all fields specified will cause the protocol to assign the associated address to the one **device** that matches all the fields (excluding address), if the **device** exists.

If several **devices** in the **system** have the same manufacturer and model numbers, and also have a serial number of ASCII character "0," the protocol can still detect the **devices** if each **device** has been previously set to a different address.

A higher level protocol in the **controller** may interact with the Set Address protocol. It may determine whether to assign ascending addresses, assign addresses that duplicate some previous configuration, or leave the addresses unchanged. It also may check the Set Address protocol output table for address conflicts. Two or more address-configurable **devices** may be at the same address or an address-configurable **device** may share the same address with a non-address-configurable **device**. The higher level protocol must decide how to resolve the conflicts and reinvoke the Set Address protocol to change the address(es) as required.

17.7.7 Standard Compliance

The SETADD common **controller** protocol may optionally be supplied by **controllers**.

17.8 Test System Protocol

17.8.1 Keyword

TESTSYS

17.8.2 Purpose

The purpose of this protocol is to evaluate the readiness of a test **system** as can be determined from the results of the self-test query, see 10.38.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

17.8.3 Information Requested by the Protocol

The protocol requires a list of addresses representing the **devices** to be self-tested.

17.8.4 Information Supplied by the Protocol

A value of zero shall be returned if every **device** passes its self-test. Otherwise, a nonzero value shall be returned that indicates the number of **devices** that have failed followed by the addresses of those **devices** and their associated *TST? query responses.

17.8.5 Controller Algorithm

```
BEGIN Self-Test System
   Initialize FAILED DEVICES quantity to zero
   Execute SEND to all of the addresses, with *TST? as
      the <PROGRAM MESSAGE> and NL^END
      as the <PROGRAM MESSAGE TERMINATOR>.
   Send the IEEE 488.1 Unlisten command (UNL)
   FOR each address in the input list
      BEGIN
         Execute RECEIVE control sequence
         IF the RECEIVE response is nonzero
            THEN
            BEGIN
               Put address and response into output table
               Increment FAILED DEVICES quantity
            END
      END
   Place FAILED DEVICES quantity into output table
END Self-Test System
```

17.8.6 Additional Requirements and Guidelines

None.

17.8.7 Standard Compliance

The TESTSYS common **controller** protocol may optionally be supplied in a **controller**.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A Compound Headers — Usage and Examples (Informative)

(These Annexes are not a part of IEEE Std 488.2-1992, IEEE Standard Codes, Formats, Protocols, and Common Commands for Use With IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation, but are included for information only.)

The use of compound headers is a relatively new concept. This appendix presents some possible techniques a device designer may consider when implementing compound headers. Several possible levels of complexity and appro aches for the device's parser are presented. Hypothetical command structures are illustrated to present some concepts device designers may find useful when designing a command structure.

A.1 Compound Header Organization Example Using a Tree

Fig A-1 presents the command structure of a prototypical device that has implemented compound headers using a tree structure. Headers were chosen with minimal mnemonic value to ensure generality.

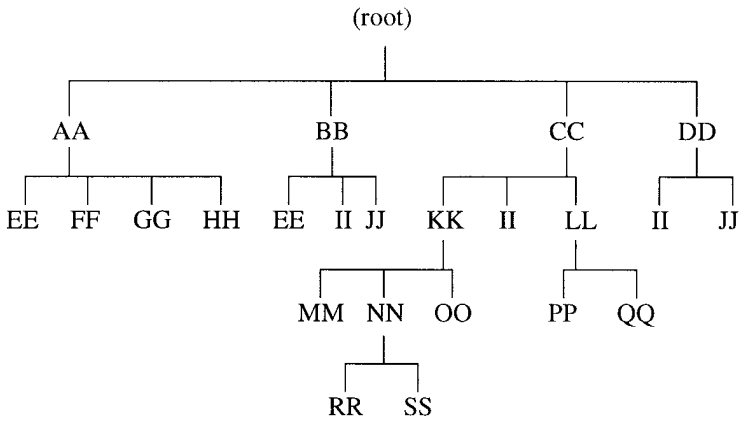


Figure A-1—Compound Header Organization Example Using a Tree

The following are some of the notable aspects of this organization:

1) The paths through the tree are not all the same length.
2) The number of subnodes under a node is not constant.
3) Node names are reused.

Some legal program commands for this hypothetical device would be

```
:AA:EE 5 <PMT>
:BB:EE 7 <PMT>
:AA:HH ON <PMT>
:CC:KK:MM 52 <PMT>
```

NOTE — <PMT> is used to indicate a <PROGRAM MESSAGE TERMINATOR>.

A.1.1 Use of the Compound Header Rules

The rules specified in 7.6.1.5 allow several implementations for the way in which a parser may behave if the leading colon on a <COMMAND PROGRAM HEADER> is omitted.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The following command sequences illustrate the workings of the Section 7. rules for command-path generation:

1) :AA:EE 5;:BB:EE 7 <PMT> The leading colon in the second <PROGRAM MESSAGE UNIT> puts the parser at the top of the command tree. Both paths are legal.
2) :AA:EE 5 <PMT> BB:EE 7 <PMT> The first <PROGRAM MESSAGE TERMINATOR> puts the parser at the top of the command tree. A leading colon at the beginning of the next <PROGRAM MESSAGE UNIT> is unnecessary because the first <PROGRAM MESSAGE UNIT> in a <PROGRAM MESSAGE> starts the parser at the root.
3) :AA:EE 7; FF 5; GG 8; HH 10 <PMT> The entire path is not given in the second, third, and fourth <PROGRAM MESSAGE UNIT> elements. The rules specify that a ":AA:" is assumed to be prefixed to the following <PROGRAM MESSAGE UNIT> elements. This command is equivalent to
AA:EE 7;:AA:FF 5;:AA:GG 8;:AA:HH 10<PMT>
4) :CC:KK:MM 3; OO 26; NN:SS 187 <PMT> The second and third <PROGRAM MESSAGE UNIT> elements are assumed to be prefixed by the implied prefix of the immediately previous command, ":CC:KK:." This command is equivalent to
:CC:KK:MM 3;:CC:KK:OO 26;:CC:KK:NN:SS 187 <PMT>
5) :AA:EE 5;*EAR 32<PMT> The processing of common commands is unaffected by any previous compound commands.
6) :AA:EE 7;*ESR 32; FF 5; GG 8; HH 10 <PMT> The insertion of the common command has no effect on the application of the prefixing rules. This command is equivalent to

:AA:EE 7;*ESR 32;:AA:FF 5;:AA:GG 8;:AA:HH 10<PMT>

The following examples would cause an error with a parser that followed only the basic rules of Section 7.:

1) :AA:EE 7; BB:EE 7<PMT> The node "BB" is not a sub-node of "AA." This type of parser is unable to look at any nodes closer to the root than node "AA."
2) :CC:KK:MM 3; LL:PP 7 <PMT> The second <PROGRAM MESSAGE UNIT> would cause an error, since node "LL" is not a subnode of "KK" The device attempts to execute the command :CC:KK:LL:PP 7, which is illegal.

A.1.2 Enhanced Tree Walking Implementation

A parser can be constructed that performs additional tree walking. The example presented here illustrates a parser that is able to "walk backwards up the tree."

This type of parser would not generate errors on the following command sequences:

1) :CC:KK:MM 3; LL:PP 7 <PMT> The parser recognizes that node "LL" is not a subnode of node "KK" The parser then looks at a level closer to the root to see if node "LL" is a subnode of node "KK"s parent.
This sequence would be interpreted the same as
:CC:KK:MM 3;:CC:LL:PP 7 <PMT>
2) :AA:EE 7; BB:EE 7 <PMT> The parser is able to recognize that nodes "AA" and "BB" have the same parent node, the root.
3) :CC:KK:MM 3; DD:II 7 <PMT> The parser recognizes that node "DD" is not a subnode of nodes "KK" or "CC." Searching continues towards the root and the parser recognizes that nodes "CC" and "DD" have the same parent, the root. The parser is able to search towards the root until it finds a match or reaches the root of the tree.
The sequence would be interpreted the same as

:CC:KK:MM 3;:DD:II 7 <PMT>

The following examples would cause an error with both types of parsers:

1) :AA:II 7 <PMT> Node "II" is not a subnode of "AA." A Command Error would be generated.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

2) :CC:KK:MM 3; EE 7 <PMT> Node “EE” is not a subnode of node “KK” nor node “CC.” Backward searching must stop at the root. A Command Error would be generated.
3) :CC:KK:MM 3 <PMT> LL:PP 7 <PMT> When the parser recognizes the first <PROGRAM MESSAGE TERMINATOR>, it will start its next search from the top of the tree. Node “LL” is not a subnode of the root. A Command Error would be generated.

A.2 Compound Header Organization Example Using a Graph

Command structures other than trees are also possible using compound headers. Fig A-2 presents a “command graph” of another prototypical device with compound headers.

Some legal program commands for this device include

:AA:BB:CC 82<PMT>
:BB:AA:CC 82<PMT>

Another graph style structure is given in Fig A-3.

Some legal program commands for this device include

:AA:BB:DD 34<PMT>
:CC:DD 34<PMT>

The device designer has the freedom to deal with complicated command structures using a great deal of creativity. The device designer should be aware that the semantics of such constructs may not be obvious to the user.

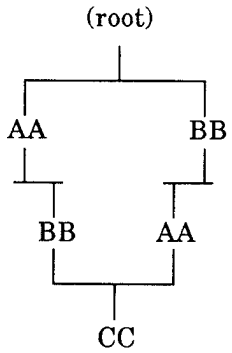


Figure A-2—Compound Header Organization Using a Graph — Example 1

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

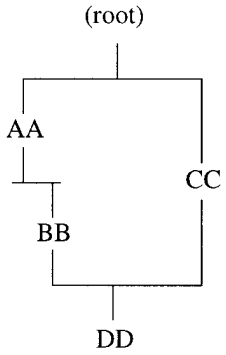


Figure A-3—Compound Header Organization Using a Graph — Example 2

A.3 Default Nodes

The command structure may be designed so that some nodes appear as defaults to the programmer. The device designer shall avoid making defaults that cannot be uniquely resolved. Even though the specific example shown in Fig A-4 is a tree structure, default nodes may be used in any valid structure as long as the requirement of unique resolution is met.

The bracketed nodes, AA, CC, KK, and NN, are default nodes in the tree structure shown in Fig A-4. The commands :AA:GG and :GG are equivalent. The longer path may provide more mnemonic value while the shorter path is easier to enter in a program.

Default nodes appear at several levels in a sequence. The command :SS is equivalent to :CC:SS, :CC:KK:NN:SS, :CC:KK:SS, :CC:NN:SS, :KK:NN:SS, :KK:SS, and :NN:SS.

NOTE — The device designer may make more than one node at a level a default node as long as any valid command will be uniquely resolved. If node BB in Fig A-3 had also been made a default node, this restriction would be violated because the valid command :II could be resolved as either :BB:II or :CC:II.

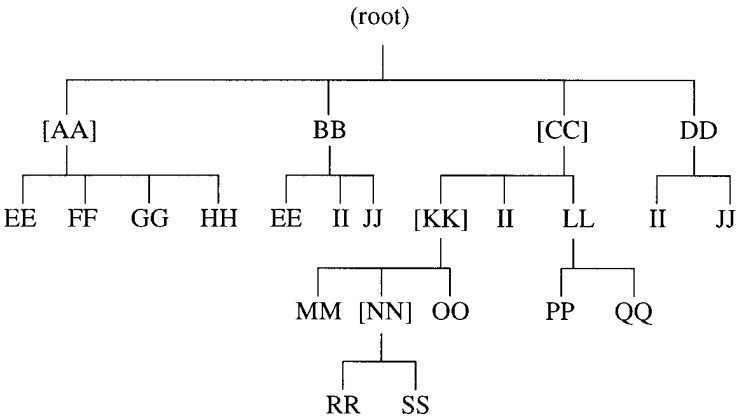


Figure A-4—Default Node Graph Example

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
--`,,`,,,-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B Device/Controller Synchronization Techniques (Informative)

This appendix presents a series of examples that illustrate the intended use of the various synchronization capabilities presented in Section 12.

This appendix contains no new **device** requirements. It presents examples and techniques that can be used in application software to ensure correct synchronization of the controller with the **device**. These examples will assist the **device** designer in understanding the functional capability provided by the **device** requirements presented in Section 12.

B.1 Simple Device and Application Program Synchronization

The IEEE 488.2 status reporting capability provides a simple means to synchronize the operation of a device with an application program. The synchronization is achieved by using one of the status reporting facilities to determine when the required **device** operation is complete.

For example: The application program sends a query command to the **device.** The query command used will be one that does not allow the device to return data until a desired operation is complete. The application program sends the query command when it must wait for the device to respond. When the **device** is ready, it places the output message in the Output Queue and sets the Message Available summary bit in the status byte. When the **controller** reads the response from the query command, the application program can continue knowing that the **device** has completed the desired operation.

Synchronization can also be achieved by using the “Operation Complete” message in the Standard Event Status Register or by using the *WAI common command.

The following examples illustrate the use of each of the following synchronization tools:

1) The *OPC? Common Query Command
2) The *OPC Common Command
3) The *WAI Common Command
4) The MAV message in the Status Byte Register

B.2 Types of Devices

For the purposes of the examples in this appendix, **devices** will be of two types, stimulus-devices or response-devices, as defined in 12.8.3.1 and 12.8.3.2.

B.2.1 Hypothetical Stimulus-Device

For the purposes of the examples in this section, a Signal Generator (SIGGEN) will be used as a hypothetical stimulus-device. It will be assumed that the IEEE 488.1 address of the **device** is represented by the decimal number 19.

The **device** is assumed to meet all the requirements of this standard. The following list of device-specific commands is assumed to exist in the device:

SETUP	This <PROGRAM MESSAGE UNIT> is assumed to do all the setup programming that the application programmer needs. In a real device , the setup might be a string of device-defined commands.
FREQ <NRf>	This <PROGRAM MESSAGE UNIT> causes the SlGGEN to generate the output frequency that is specified in the NRf numeric format.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

FREQ:INC — This <PROGRAM MESSAGE UNIT> causes the SIGGEN to increment its output frequency by a fixed amount.

NOTE — The device-specific headers used by this hypothetical device have been selected to make the following examples understandable. They are not intended to set a precedent for real devices.

B.2.2 Hypothetical Response-Device

For the purposes of the following examples, a Digital Volt Meter (DVM) will be used as a hypothetical response-device. It will be assumed that the IEEE 488.1 address of the **device** is represented by the decimal number 8.

The device is assumed to meet all the requirements of this standard. The following list of device-specific commands is assumed to exist in the device :

SETUP — This <PROGRAM MESSAGE UNIT> is assumed to do all the setup programming that the application programmer needs. In a real device, the setup might be a string of device-defined commands. SETUP is a Sequential Command.

MEAS — This <PROGRAM MESSAGE UNIT> causes the DVM to initiate the measurement.

MEAS? — This <PROGRAM MESSAGE UNIT> causes the DVM to initiate the measurement and return a <RESPONSE MESSAGE> upon completion of the measurement operation.

LAST? — This <PROGRAM MESSAGE UNIT> causes the DVM to return the results of the last measurement without triggering a new one. The measurement will be returned as a single <RESPONSE MESSAGE>.

MEAS:CONT? — This <PROGRAM MESSAGE UNIT> causes the DVM to take a measurement and return its results as a <RESPONSE MESSAGE UNIT>. After the controller reads the measurement result, the DVM takes another measurement and provides its results in the same manner. The process continues as long as the controller will accept the data. The <RESPONSE MESSAGE UNIT> elements are separated by <RESPONSE DATA SEPARATOR> elements, i.e., a comma. The <RESPONSE MESSAGE TERMINATOR> is not sent.

TRIG:EXT — This <PROGRAM MESSAGE UNIT> configures the device such that it will not perform a measurement until an External Control Signal is stimulated. When the signal is stimulated, the DVM will take a measurement.

*DDT #0 … — This is an optional common command that is used to determine the function of a Group Execute Trigger. It is defined in 10.4.

NOTE — The device-specific headers used by this hypothetical device have been selected to make the following examples understandable. They are not intended to set a precedent for real devices .

B.2.3 Hypothetical Controller Language

For the purposes of the examples in this section, a hypothetical controller that has the following language will be used. The keywords from Sections 16. and 17. have been used.

RESET a — An Interface Clear (IFC) is performed. A Device Clear (DCL) is sent. The *RST common command is sent to the device whose IEEE 488.1 address is represented by the number“a.”

SEND a; “ ” — The number “a” will be used as the IEEE 488.1 address. Data enclosed in the quotes will be sent to the device . The sent message will be automatically terminated using NL^END.

RECEIVE a; num — The number “a” will be used as the IEEE 488.1 address. A <RESPONSE MESSAGE UNIT> is read from the device . The numeric results are placed in the variable “num.”

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

TRIGGER al, a2, ... , an	A Group Execute Trigger command (GET) will be sent to all devices whose IEEE 488.1 address is in the list of numbers “al” through “an.”
READ STATUS BYTE a; sts	The number “a” will be used as the IEEE 488.1 address. The IEEE 488.2 status byte is read from the device using a serial poll. The numeric result is placed in the variable “sts.”
WAIT SRQ	The controller waits for a SRQ on the IEEE 488.1 bus. During the waiting time, the Application Programmer can use the controller to do other useful work. The details of how the controller proceeds doing useful work while at the same time waiting for a SRQ will not be shown.

REPEAT
BEGIN
END
	Statement between BEGIN and END will be repeated indefinitely.

IF (descriptive expression)
THEN
ELSE
	If the descriptive expression is TRUE, statements after THEN will be processed. IF it is FALSE, statements after ELSE will be processed. BEGIN and END may be used to group statements after a “THEN” or an “ELSE.”

B.3 Synchronization With Stimulus-Devices

Stimulus-devices generally require device-defined setup commands to specify the desired output signal. The application program must then determine that the stimulus is valid before it allows another component in the system to utilize the signal. This synchronization can be done with the *OPC? query command. The “Operation Complete” query permits a device to indicate when it has completed an operation that was initiated by a <PROGRAM MESSAGE UNIT>.

A stimulus-device will indicate that it has completed a stimulus operation only when its outputs are valid.

For example, a stimulus-device is sent a programming message to change its output followed by an *OPC? query command. The Operation Complete message is placed in the Output Queue when the device has recognized the *OPC? query command and has also provided the output signal specified by the previous programming message.

B.3.1 Stimulus-Device Synchronization Using a <RESPONSE MESSAGE>

The following example illustrates the simplest method for synchronizing a stimulus-device with an application program. In this example, the application program uses device-specific commands to specify the desired stimulus. The *OPC? query command is used to provide a <RESPONSE MESSAGE> when the output signal is valid.

The principles of this example apply equally well to device functions that do not provide a stimulus, but do take time and do not return a <RESPONSE MESSAGE> Note that, in the following example, the controller will not be able to complete the read operation in STEP 5 until the device's output has changed to the new frequency This will cause the IEEE 488.1 bus to be held-off while the controller is waiting for the device The read operation is used for synchronization only; the numeric value read is ignored.

Stimulus Synchronization Using a <RESPONSE MESSAGE>

STEP	
(1) RESET 19	Send IFC, DCL, *RST.
(2) SEND 19; “*SRE 0”	Turn off all Service Requests.
(3) SEND 19; “SETUP”	
(4) SEND 19; “FREQ 10.0E3; *OPC?”	Set Frequency to 10 kHz.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

(5) RECEIVE 19; num — The number returned and stored in variable "num" is an ASCII character "1," discard it.

… proceed with application, knowing that the device's
…new output is valid.
…
…

B.3.2 Stimulus-Device Synchronization With Service Request

The following example illustrates how a stimulus-device can be synchronized with an application program by using SRQ. In this example, the device is programmed to generate a new output signal and generate a service request when the new stimulus signal is valid Upon receiving the service request, the controller programs other elements in the system to make use of the stimulus signal.

The device designer will ensure that the SRQ will not be generated until the desired output signal is valid.

The controller can perform other useful work while the device is changing its output.

The application program sends the device-specific stimulus command "FREQ" followed by the *OPC? common query command. The device sets the Message Available Summary bit in the status byte when the stimulus is valid. Note that proper use of the Message Available Summary bit in the status byte eliminates the need for application program time delays

Stimulus Synchronization With Service Request

STEP

(1) RESET 19 — Send IFC, DCL, *RST.
(2) SEND 19; "SETUP"
(3) SEND 19; "*SRE 16" — Enable MAV Service Request.

(4) SEND 19; "FREQ 10E3; *OPC?" — Set Frequency to 10 kHz.

(5) WAIT SRQ — While waiting for Service Request **controller** can do other useful work.

(6) READ STATUS BYTE 19; sts

(7) IF (MAV message is NOT in sts)
THEN — unexpected SRQ!
BEGIN
..process unexpected SRQ
GO TO 5
END

(8) RECEIVE 19; num num is an ASCII character "1," discard it.

… direct the system to do something useful
… with the stimulus signal.

B.3.3 Stimulus-Device Synchronization Without <PROGRAM MESSAGE> Elements

The following is an example of how a stimulus-device can be operated without requiring <PROGRAM MESSAGE> elements after the initial setup.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Assume that the device is the hypothetical Signal Generator (SIGGEN) defined above. Assume also that it can have its output frequency incremented by use of the IEEE 488.1 Group Execute Trigger message (GET).

In this example, the output frequency of the SIGGEN is incremented each time a GET is received. The Message Available bit (MAV) in the status byte is read via a serial poll so that the controller will know when the new frequency is valid.

Each time a GET is sent to the SIGGEN, it is the same as sending the <PROGRAM MESSAGE> "FREQ:INC; OPC?." The new program message empties the Output Queue, which causes MAV to go FALSE. When the operation is complete, the *OPC? response, an ASCII character "1," will be placed into the Output Queue. Since the Output Queue is not empty, MAV becomes TRUE, which in turn causes the device to make SRQ TRUE.

The new <PROGRAM MESSAGE> also causes the device to report a query error as the unread contents of the Output Queue are discarded when a new <PROGRAM MESSAGE> is received. This error is ignored as the ESB bit is not enabled by the *SRE 16 command.

Stimulus-Device Synchronization Without <PROGRAM MESSAGE> Elements

STEP	
(1) RESET 19	Send IFC, DCL, *RST.
(2) SEND "*SRE 16"	Enable (MAV) Service Request.
(3) SEND 19; "SETUP; FREQ 1000"	Initialize Frequency. Assume that SETUP includes setting the increment size that will be used each time a GET is received
(4) SEND 19; "*DDT #0 FREQ:INC;*OPC?"	Causes **device**· to respond as though "FREQ:INC; OPC? NL^END" had been received each time a GET is received. Device is now ready to Accept GET mes sages.
(5) REPEAT	
(6) BEGIN	
(7) TRIGGER 19	**Controller** sends GET here. The SRQ will not occur until after the GET is received and the SIGGEN has settled at the next frequency.
(8) WAIT SRQ	While waiting for Service Request, ·**controller** can do other useful work.
(9) READ STATUS BYTE 19; sts	
(10) IF (MAV message is NOT in sts)	
(11) THEN	Unexpected SRQ from another IEEE 488.1 **device**
(12) BEGIN	
.. process unexpected SRQ!	
(13) GO TO 8	
(14) END	

... direct the ·**system** to do something useful
... with the stimulus signal.

(15) END

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The application performs the following in a loop:

STEP:

1) Controller sends GET to Signal Generator
2) Wait for Service Request
3) Perform a serial poll on the device

Since the GET message does the equivalent of a <PROGRAM MESSAGE> after a prior query command was terminated, the Output Queue is discarded and, consequently, the MAV message goes FALSE. Discarding the Output Queue causes a Query Error.

B.4 Synchronization With Response-Devices

Response-devices generally require device-defined setup commands and a query command followed by the controller reading the resultant measurement The following examples illustrate this method

B.4.1 Simple Response-Device Synchronization

The following example illustrates the simplest method for synchronizing a response-device with an application program. In this example, the application program requests a measurement and then waits at step 7 for the results. The sequence is repeated indefinitely The read measurements are always valid because the device does not send data until it has completed each measurement.

Response-Device Synchronization Using a <RESPONSE MESSAGE>

```
STEP
  (1) RESET 8                    Send IFC, DCL, *RST.
  (2) SEND 8; "*SRE 0"           Turn off all Service Requests.
  (3) SEND 8; "SETUP"

  (4) REPEAT
  (5)   BEGIN
  (6)     SEND 8; "MEAS?"
  (7)     RECEIVE 8; num
  (8)     ... do something useful with the number...
  (9)   END
```

B.4.2 Response-Device Synchronization With Service Request

The following example illustrates the Service Request method for synchronizing a response-device with an application program. In this example, the controller requests a measurement and then waits for a service request to be generated. Upon receiving the service request, the controller reads the results. The sequence is repeated indefinitely. The read measurements are always valid because the device does not issue the service request until it has completed each measurement. The controller can be used to perform other useful work while the device is performing the measurement.

The application program sends the device a device-specific query to retrieve the measurement data. The device sets the Message Available Summary bit in the status byte when the measurement data is available The Message Available Summary bit in the status byte eliminates the need for application program time delays.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Response-Device Synchronization With Service Request

```
STEP
  (1) RESET 8                          Send IFC, DCL, *RST.
  (2) SEND 8; "*SRE 16"                Enable (MAV) Service Request.
  (3) SEND 8; "SETUP"

  (4) REPEAT
  (5)  BEGIN
  (6)   SEND 8; "MEAS?"
  (7)     WAIT SRQ                     While waiting, controller can do other useful work.

        .. program continues when SRQ is received

  (8)     READ STATUS BYTE 8; sts
  (9)     IF (MAV message is NOT in sts)
  (10)      THEN                       unexpected SRQ!
  (11)        BEGIN
   .. check other devices and
   .. process unexpected SRQ
  (12)          GO TO 7
  (13)        END
  (14)   RECEIVE 8; num
        .. do something useful with the number...
  (15) END
```

B.4.3 Response-Device Synchronization Without <PROGRAM MESSAGE> Elements

The following example illustrates how a response-device can be made to take a series of measurements without <PROGRAM MESSAGE> elements being used within the loop. The application allows for quick, low-overhead reading of measurement results. In this example, the controller requests the device to return measurement results as a continuous stream of <RESPONSE DATA> elements Read measurements are always valid because the device does not send data until it has completed each measurement We assume that the controller returns comma (,) delimited measurement values, one by one, to the application program.

Response-Device Synchronization Without <PROGRAM MESSAGE> Elements

```
STEP
  (1) RESET 8                          Send IFC, DCL, *RST.
  (2) SEND 8; "*SRE 0"                 Turn off all Service Requests.
  (3) SEND 8; "SETUP; MEAS:CONT?"

  (4) REPEAT
  (5)  BEGIN
  (6)    RECEIVE 8; num                (delimited by ",")
  (7)   .. do something useful with the number...
  (8)  END
```

B.4.4 Device Communications While Waiting for a Measurement

The following example illustrates how a response-device can be programmed to take a measurement and then, while the device is performing it, allow the device to send and receive messages. This capability can be useful when a measurement is very time consuming and there is some useful information that the controller may wish to read from

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

the device while the measurement is in process. The example uses the *OPC common command to generate the Operation Complete message in the Standard Event Status Register when the measurement completes. This example also uses a service request to eliminate the need to poll the Standard Event Status Register with the *ESR? common query command.

Device Communications While Waiting for a Measurement

```
STEP
  (1) RESET 8                          Send IFC, DCL, *RST.
  (2) SEND 8; "SETUP"
  (3) SEND 8; "*SRE 32; *ESE 1"        Enable (ESB) Service Request. Allow
                                         OPC bit to set ESB message.
  (4) SEND 8; "MEAS; *OPC"             Begin a measurement.
  (5) WAIT SRQ
...
...Perform other tasks involving the device by
...sending <PROGRAM MESSAGE> elements or reading
...<RESPONSE MESSAGE> elements. Avoid any of the
...following: *CLS, *RST, *SRE, *ESE
...or any device-specific command
...that has a pending operation.

  (6) READ STATUS BYTE 8; sts
  (7) IF (RQS message is TRUE in sts)
        THEN
         BEGIN
           SEND 8; "LAST?"
           RECEIVE 8; num
           .. do something useful with the number..
         END
      ELSE GO TO "Unexpected Service Request Handling"
```

NOTE — The Standard Event Status Register does not have to be read to determine that the Operation Complete message is TRUE because this is the only message that has been enabled to generate a SRQ. A *CLS or a query of the Standard Event Status Register is required to remove the SRQ if the procedure is to be repeated.

B.4.5 Synchronization Using an External Control Signal

The synchronization of a response-device with an application program can be accomplished by use of an external-control-signal. This method may be necessary when an asynchronous event must determine when a measurement is taken. This synchronization can be accomplished using any of the three methods indicated in the previous examples. The only changes required are to add device-specific setup commands that direct the device to take measurements only when the external-control-signal is received. (See the "TRG:EXT" command for the hypothetical DVM.)

The measurements read are always valid because the device does not send data until it has completed each measurement.

B.4.6 Example Involving Simultaneous Trigger of Two Response-Devices

The following example shows how two response-devices can be commanded to take a measurement at the same time. The IEEE 488.1 Group Execute Trigger (GET) command is used to start the measurement. Both devices are assumed to be the hypothetical Digital Volt Meter (DVM) defined earlier. The two devices have IEEE 488.1 addresses of 8 and 9. The application requires that a pair of voltages be sampled at the same time.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

This example uses the *OPC? query command to verify that each device has emptied its Input Buffer and has executed all prior commands. Since the device's Input Buffer is known to be empty, the application is in a state in which it can immediately execute the Group Execute Trigger (GET) command Since the two devices are the same make and model, this example assumes that the two measurements will be taken close together in time.

If the timing needs to be controlled any closer than can be achieved with a Group Execute Trigger, then the application program will have to use means outside of IEEE 488.1 to accomplish the parallel trigger. This triggering could be done using special external trigger hardware in the **devices.**

Simultaneous Trigger of Two Response-Devices

STEP	
(1) RESET 8	Initialize (DVM-1).
(2) RESET 9	Initialize (DVM-2).
(3) SEND 8; "*SRE 0; SETUP"	Turn off Service Requests. Set up (DVM-1).
(4) SEND 9; "*SRE 0; SETUP"	Turn off Service Requests. Set up (DVM-2).
(5) SEND 8; "*DDT #0 MEAS?"	Set DVM-1 to perform a measurement upon receipt of a Group Execute Trigger.
(6) SEND 9; "*DDT #0 MEAS?"	Set DVM-2 to perform a measurement upon receipt of a Group Execute Trigger.
(7) SEND 8; *OPC?	
(8) RECEIVE 8; num	The number returned and stored in variable num is an ASCII "1," discard it. DVM-1's parser is idle.
(9) SEND 9; "*OPC?"	
(10) RECEIVE 9; num	The number returned and stored in variable num is an ASCII "1," discard it. DVM-2's parser is idle.

… At this point, the application program directs other
…components in the system to provide the signals
…that are to be measured.

(11) TRIGGER 8,9;	This command addresses DVM-1 and DVM-2 to listen and then sends the Group Execute Trigger Message.
(12) RECEIVE 8; num-1	When the measurement from DVM-1 completes, it is sent to the controller and placed in variable num-1.
(13) RECEIVE 9; num-2	When the measurement from DVM-2 completes, it is sent to the controller and placed in variable num-2.

…
… do something useful with the read
…measurement data in num-1 and num-2.

NOTES:

1 — Even though the measurements were read by the controller at different times, they were completed at the same time.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

2 — The *DDT commands would be unnecessary if the device had implemented the "DTI" subset and simply defined Group Execute Trigger to cause the device to take a measurement. If the device has been built this way, the above example might be changed in the following manner:

The application program would have placed the DVMs in a deferred trigger mode as part of the setup. This mode would keep the DVM from taking a measurement until triggered by a GET. The DVM "MEAS?" query commands would not need to be sent to each DVM at all. The GET would be sent to both devices causing parallel measurements. The resultant data would then be read from each DVM.

B.5 Generalized Synchronization Independent of Queries

Synchronization with device operation can also be done with the *OPC command This command is identical to the *OPC? query command except that the OPC bit is set in the Standard Event Status Register instead of generating the ASCII "1" as an output message. The *OPC command can be used to synchronize either stimulus-devices or response-devices. The *OPC bit of the event status register can be monitored to determine that all pending operations have been completed.

Assume a hypothetical device that can perform three overlapped operations: OP1, OP2, and OP3. Each of these commands are Overlapped Commands as defined in 12.2.

Synchronization Independent of Queries

```
STEP
  (1) RESET 19                          Send IFC, DCL, *RST.

  (2) SEND 19; "SETUP"

  (3) SEND 19; "*SRE 32; *ESE 1; *CLS"  Enable (ESB) Service Request. Allow OPC bit to set ESB message
                                        CLEAR Standard Event Status Register

  (4) SEND 19; "OP1; OP2; OP3; *OPC"    Begin a measurement.

  (5) WAIT SRQ

…
…Perform other tasks involving the device by
…sending <PROGRAM MESSAGE> elements or reading
… <RESPONSE MESSAGE> elements. Avoid any of the
…following: *CLS, *RST, *SRE, *ESE
…or any Overlapped Command
…

  (6) READ STATUS BYTE 19; sts

  (7) IF (RQS message is TRUE in sts)
        THEN
          BEGIN
          …Device has completed all three
          … commands: OP1, OP2, and OP3.
          …
          … Proceed with application.
          …
END
ELSE GO TO "Unexpected Service Request Handling"
```

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE — The Standard Event Status Register does not have to be read to determine that the Operation Complete Message is TRUE because this is the only message that has been enabled to generate a SRQ. However, before the program could be written as a loop, the Standard Event Status Register would have to be cleared each time. Clearing can be done by sending a *CLS command or by reading the Standard Event Status Register by using the *ESR? query command.

B.6 Device Synchronization Using the *WAI Command

The following example illustrates the simplest method for ensuring that device operations are performed in order.

In this example, the application program uses device-specific commands to start an operation in a response-device. The *WAI command is used to force the device to complete this operation before allowing the device to perform a measurement.

The device will be commanded to initiate an internal calibration operation with the following command:

"CALIBRATE"

For the purposes of this example, it is assumed that the "CALIBRATE" command is an Overlapped Command and starts an operation as defined in 12.2. By using the *WAI command directly after "CALIBRATE," this Overlapped Command operates as though it was a Sequential Command. The application program then commands the device to perform a measurement. Since the *WAI command was used, the application program knows that the measurement was performed after the calibration operation had been completed.

Device Synchronization Using the *WAI Command

STEP	
(1) RESET 8	Send IFC, DCL, *RST.
(2) SEND 8; "*SRE 0"	Turn off all Service Requests.
(3) SEND 8; "SETUP"	
(4) SEND 8; "CALIBRATE; *WAI"	Start Calibration, the *WAI command forces the ·**device** to complete the cali bration operation before the <PROGRAM MESSAGE UNIT> in Step 5 is executed.
(5) SEND 8; "MEAS?"	The ·**device**· is commanded to take a measurement and return its results as a <RESPONSE MESSAGE>.
(6) RECEIVE 8; num	The number returned and stored in variable num is the valid measurement results.

NOTE — If the *WAI command had not been used, steps (5) and (6) would have been completed before the calibration operation had finished. Therefore, the measurement would not have been performed at the correct time.

B.7 System Example Involving Both a Response-Device and a Stimulus-Device

The following example involves a system with four components:

1) An IEEE 488.2 stimulus-device (hypothetical SIGGEN)
2) An IEEE 488.2 response-device (hypothetical DVM)
3) An IEEE 488.2 system controller (hypothetical language)
4) A two-port device under test (DUT)

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The system topology is illustrated in Fig B-1.

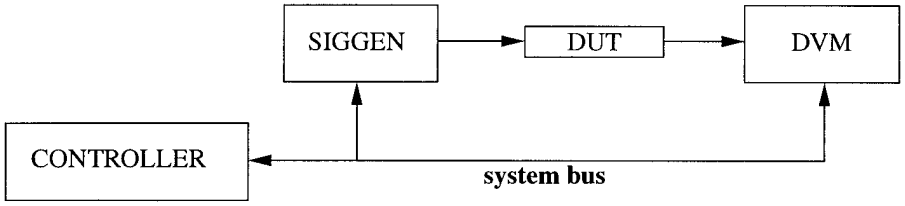


Figure B-1—System Topology for Response-Device and Stimulus-Device Example

The application calls for a series of measurements to be made where the SIGGEN stimulates the DUT with 1000 different frequencies. The DVM measures the DUT response at each frequency and sends the data to the controller .

The IEEE 488.1 address for the SIGGEN is 19. The address for the DVM is 8.

The following example is very simple and does not use any Service Requests.

Step

(1) RESET 8	Send IFC, DCL, and *RST to initialize (DVM).
(2) RESET 19	Send IFC, DCL, and *RST to initialize (SIGGEN).
(3) SEND 8; "*SRE 0; SETUP"	Turn off Service Requests. Setup (DVM).
(4) SEND 19; "*SRE 0; SETUP"	Turn off Service Requests. Setup (SIGGEN).
(5) SEND 19; "FREQ 1000; *OPC?"	Set SIGGEN Start Frequency.
(6) REPEAT 1000 TIMES	
(7) BEGIN	
(8) RECEIVE 19; num	Response is ASCII "1," ignore.
(9) WAIT for DUT response time	
(10) SEND 8; "MEAS?"	Command (DVM) to measure.
(11) RECEIVE 8; num	Read from (DVM).
…	
…do something useful with the read	
…measurement data	
…	
(12) SEND 19; "FREQ:INC; *OPC?"	Increment SIGGEN Frequency.
(13) END	

NOTE — The WAIT statement in step (9) is to allow time For the DUT to respond to the stimulus signal that was applied. It is not necessary to allow for timing in the SIGGEN or DVM. Their timing requirements are taken care of by correct implementation of the synchronization features of this standard.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex C Automatic System Configuration Example (Informative)

The following example illustrates the interaction of the SETADD protocol and the *AAD common command in detecting three address-configurable devices on a IEEE 488.1 bus. The identifiers for the three devices are shown in Table C-1, along with the addresses that will be assigned. Each identifier consists of the four fields shown.

The protocol searches out each character (a data byte on the IEEE 488.1 bus) of each identifier from left to right (manufacturer to current listen address) and will conduct the search for as many times as there are unique identifiers.

Table C-1—Device Identifiers/Assigned Addresses

	|< ..... Identifiers ..... >|			
Device	**Manufacturer**	**Model #**	**Serial #**	**Current Address**	**Assigned Address**
#1	N	B	4	! (1)	! (1)
#2	N	BB	3	$ (4)	% (5)
#3	M	A	6	! (1)	# (3)

C.1 Overall Flow of the Protocol

The **devices** are detected in the order shown in Fig C-1. Each identifier search is initiated with the STX character (decimal value of 2).

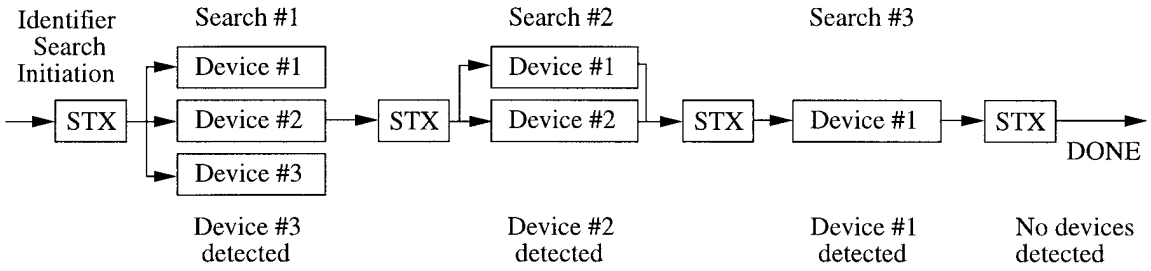


Figure C-1—Overall Protocol Flow

Device #3 is detected first because the protocol chooses the identifier that has the lowest-valued character in the position it is testing. In this example, the “M” appears first as the lowest-valued character. Its decimal value is 77, whereas “N” has a value of 78. Device #2 is detected next because the protocol chooses a superset of a field over that field itself. That is, the superset field “BB” is chosen over “B.” Finally, only Device #1 is left and is guaranteed to be detected since it has exclusive control in guiding the controller to its identifier characters.

C.2 Description of the Identifier Searches

The three searches are described in more detail in Fig C-2. The participation profile of each device is given along with the identifier characters that have been determined by the controller. <eof> indicates that an end-of-field condition has occurred. The controller detects <eof> when SRQ is found to be FALSE after the byte with a value of 127 is sent. A succession of two <eof>'s is a signal that the entire identifier has been searched and a device has been detected (only one still participating).

--`,,`,,,-`-`,,`,,`,`,,`---
Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

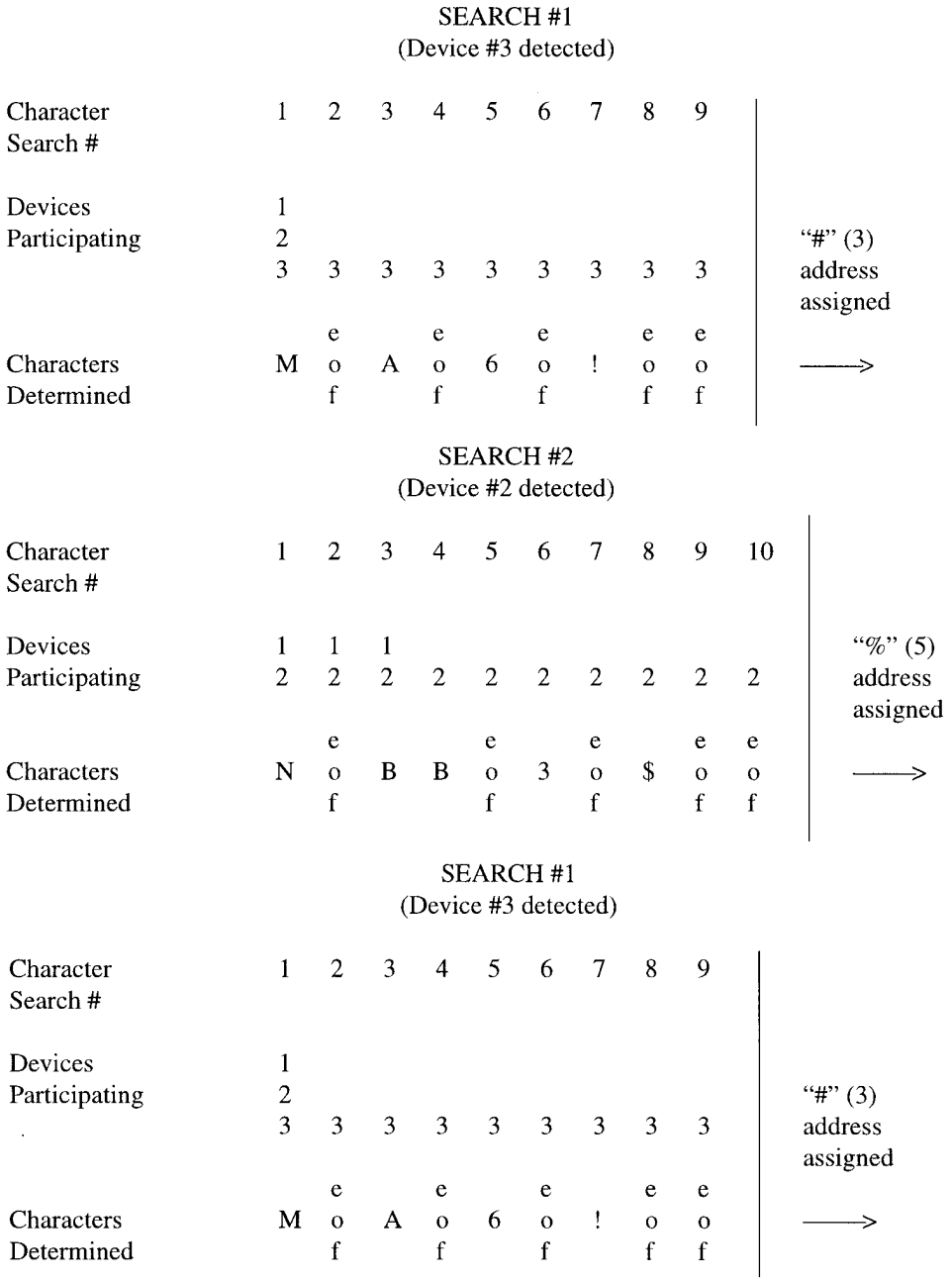


Figure C-2—Identifier Searches

C.3 Description of the Character Search

Each character search is made up of a character search initiation character and a character search body, Fig C-3.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

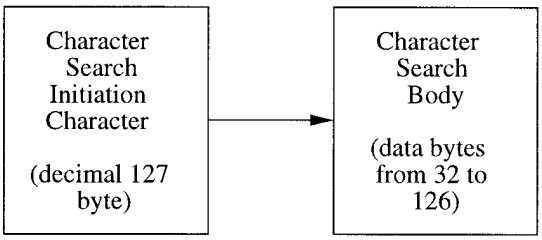


Figure C-3—Character Search Elements

C.3.1 The Search Initiation Character

A participating device is required to assert SRQ TRUE if the controller places a data byte on the bus whose value is greater than the identifier character it is using for comparison. Otherwise, the device is required to assert the SRQ line passive FALSE. Since the maximum value that an identifier character can have is decimal 126, the Character Search Initiation Character, decimal 127, will cause any device desiring a character to be searched to assert SRQ TRUE. This action will cause the controller to execute the search body.

C.3.2 The Search Body

The search body consists of a seven-step successive-approximation binary search that starts with a data byte having a decimal value of 64 (the "@" character) and a delta of value 32. If the data byte causes an SRQ indication, the **controller** subtracts the delta from the data byte to form the next data byte. Otherwise, it adds the delta. It then halves the delta. This process continues until the data byte calculated with the delta of I is sent. From the resulting SRQ indication, the **controller** can determine the lowest-valued character active in the search. If the SRQ line is FALSE, the character is equal to the last data byte sent. If it is TRUE, the character is one less than the last data byte sent. At this point, the **controller** sends the character it has determined with the DI08 line asserted TRUE. (Previously, DIO8 was FALSE.) This line asserted TRUE indicates to each participating **device** that if its character equals the data byte (excluding the DIO8 value), then the **device** is to continue participating. Otherwise, the **device** is to cease participating until the next STX character signals the beginning of a new identifier search. A graphical representation of this process is shown in Fig C-4.

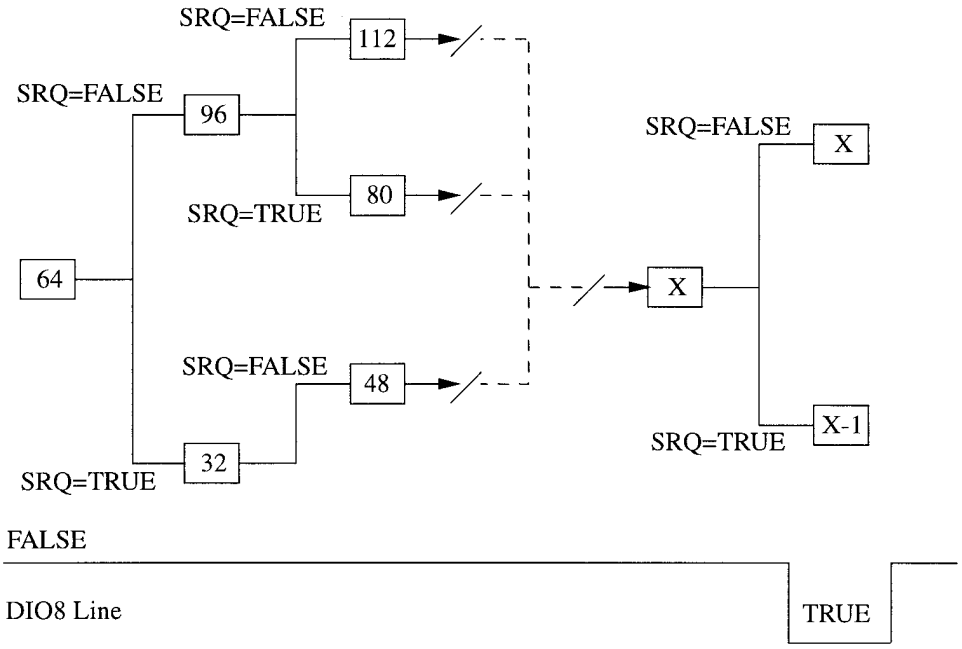
C.4 Detailed Description of the Search Process

C.4.1 Manufacturer Search for the Three Sample Identifiers

Fig C-5 illustrates the searching sequence for detecting the "M" character of Device #3. The SRQ lines of the active **devices** interact to guide the controller to the lowest-valued character being presented.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



NOTES: (1) Each box represents a data byte sent by the **controller** and its decimal value is shown within.
(2) "X" equals the data byte calculated with delta equal to one. Thus, X or X-1 represents the lowest-valued character, depending upon the SRQ line state following transmission of "X."

Figure C-4—Search Body

The lowest-valued character "M" (value 77) caused no SRQ TRUE indication with the delta of 1. Thus, it was sent again with DIO8 TRUE. This action caused Devices #1 and #2 to cease participating in the protocol because their characters did not equal "M."

Immediately after the "M" is detected, the figure shows that an end-of-field (<eof>) was detected. That is, Device #3 did not assert SRQ TRUE upon receiving the next character search initiation character (decimal 127) in step 10. Had Device #3 been at the end of its last field, it would again indicate an <eof> condition upon receiving the character search initiation character to indicate a zero length field. The controller is then aware that a **device** has been detected.

C.4.2 Detection of a Superset Field

Fig C-6 illustrates the manner in which the protocol selects the superfield. The first "B" is detected in steps 1-9. Then, the controller sends the next search initiation character in step (10). Device #1 responds by asserting SRQ passive FALSE as an indication that <eof> has been reached. Device #1 then expects the controller to send another search initiation character to start the search of its serial number field. However, the **controller** detected an active TRUE SRQ assertion from Device #2 and starts the search body with a decimal 64 data byte. As a result of not seeing the second decimal 127 data byte, Device #1 ceases participating.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

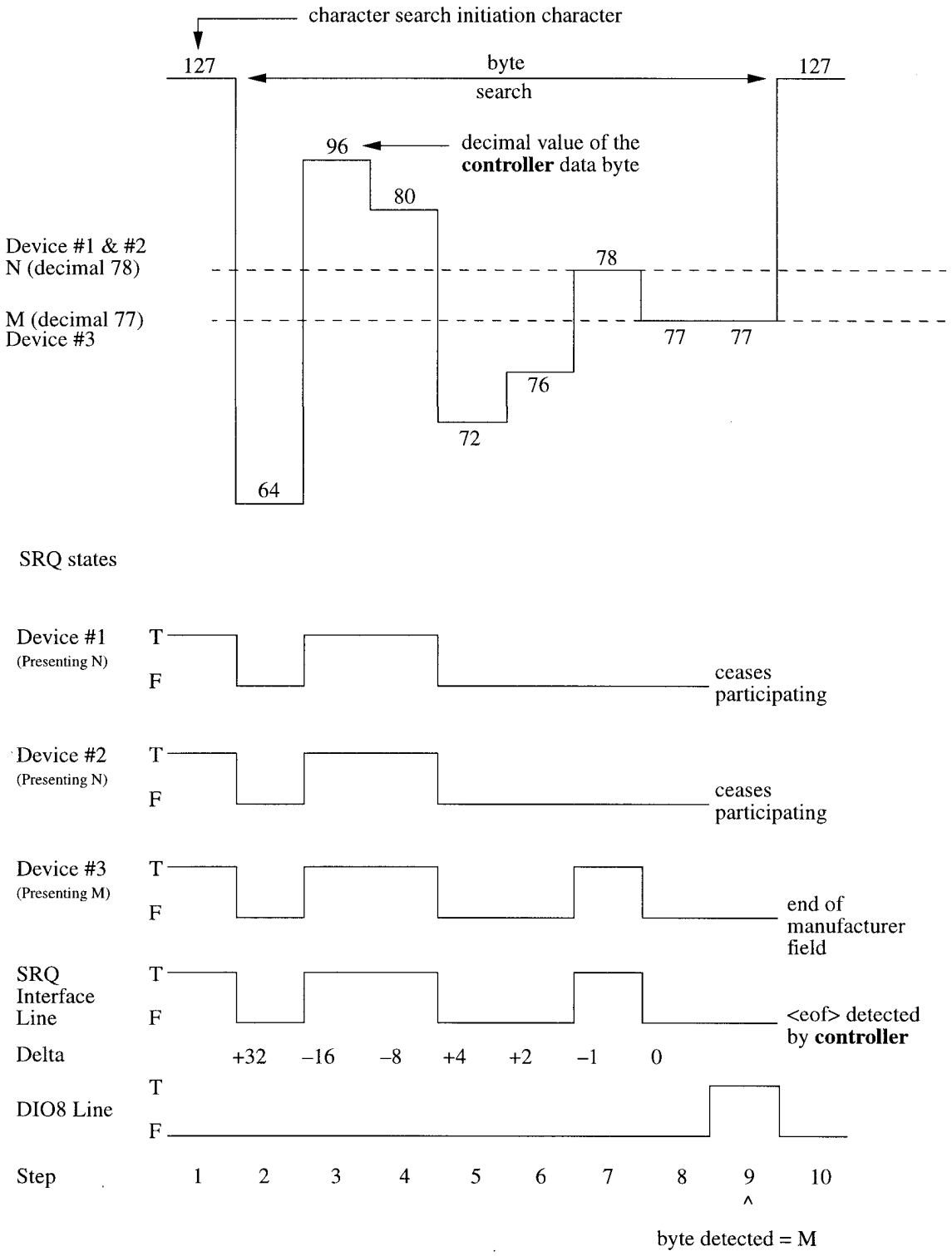


Figure C-5—Character Search Action for a Manufacturer Character and End-of-Field <eof>

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

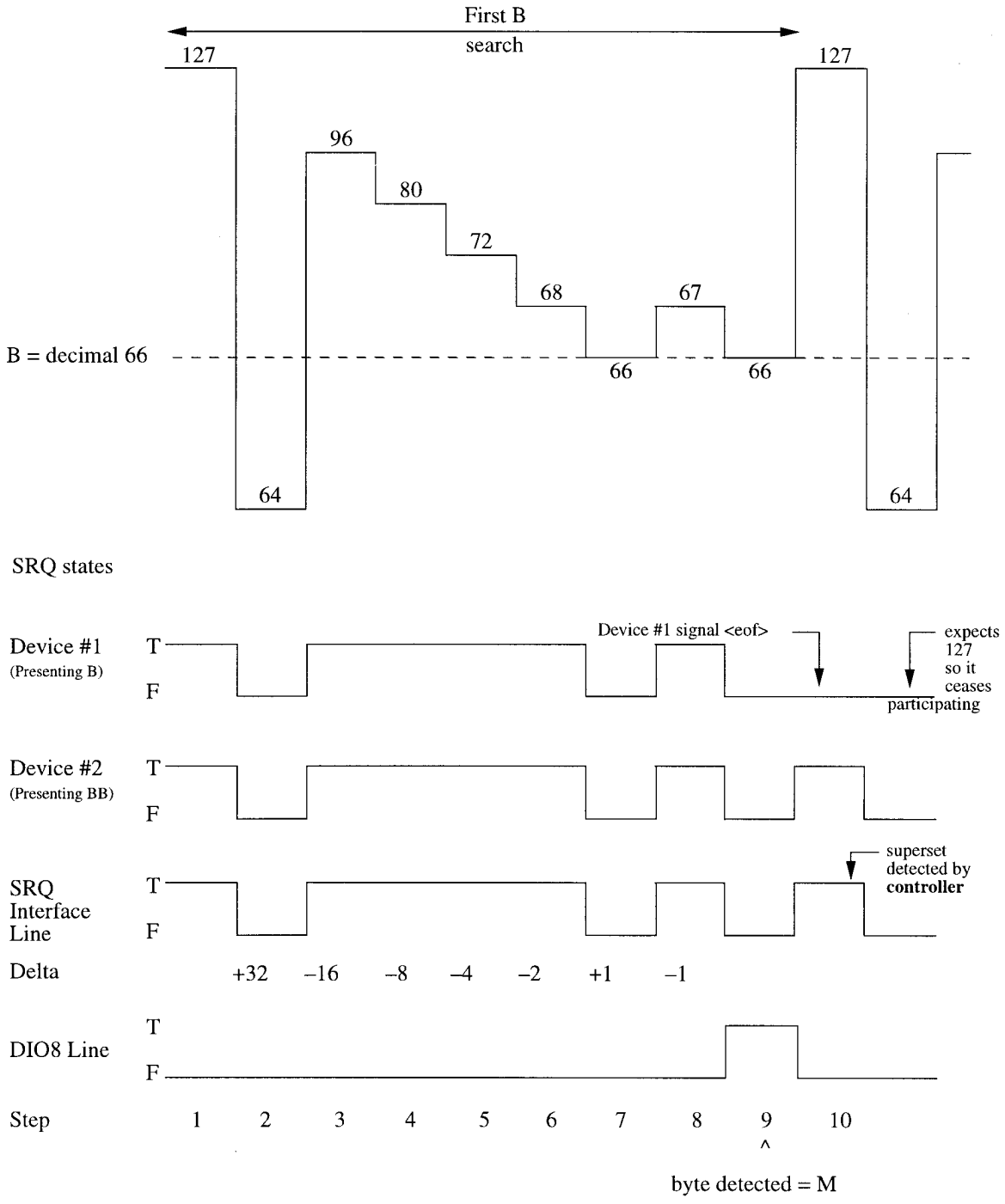


Figure C-6—Detection of the Superset Field BB

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex D Reset Guidelines (Informative)

Several events can occur that cause a device function to enter a specific default state. Besides the common commands *RST and *CLS, the interface messages dcas and IFC, or a power-on event, also may cause particular device functions to become reset and enter a specific default state. The reset requirements, as specified throughout this standard, depend both upon the class of a function and its hierarchical level. This appendix defines several device functional classes and levels and then presents the overall reset strategy in a set of tables using this classification system. Tables D-1 through D-4 summarize the specific reset actions of *RST, dcas, *CLS, and power-on respectively. Table D-5 summarizes all reset actions.

This standard also specifies documentation requirements relating to resetting. See 4.9 for specific information on these requirements.

D.1 Definitions

D.1.1 Device Functional Class Definitions

device settings: Device conditions that can be specified typically by program message units. Device settings do not depend upon the progress of **device** processes. This means that they do not depend upon the successive states of these processes. **Device** settings remain unchanged and available within the **Device** until otherwise specified. **Device** settings affect (condition) the **device** (measuring) process rather than control these processes (control = specify action, initiate process). In general, a value can be assigned to a **device** setting, such as on/off, 1 through 5, etc.

current device settings: Device settings that are currently in use. When a **device** is equipped with a nonvolatile memory, they are the settings that are saved in such memory (see also **memorized device settings**).

memorized device settings: Device settings that are saved in some memory location that do not affect current **device** processes (e.g., measuring process). However, memorized **device** settings need to be recalled to become the current **device** settings.

protected settings: Device conditions that cannot be changed by remote program messages unless particular, specific measures are taken. Protected settings remain unchanged during power-off (e.g., calibration data, user-defined data, etc.).

device states: Device conditions that typically reflect the states of **device** processes (e.g., the execution phase of a measuring process). They change with the progress of the **device** processes and are not explicitly specified by program messages. A **device** state may be the result (directly or indirectly) of a (process) control command.

operation modes: Device conditions that have the same characteristics as current **device** settings, except that they would not be saved in a nonvolatile memory location. For example, suppose a device is operating in a particular scan mode. When the settings are saved and then recalled later, the scan mode appears to be switched off because it was not saved. In that case, the scan mode is identified as an operation mode. Whenever the scan mode is not switched off after a recall, it is identified as a device setting.

configuration settings: Settings that describe the hardware and software configuration and its characteristics. Configuration settings typically are not changed during execution of an application program. Configuration settings are mostly fixed, manually selectable by hard local controls (such as dip switches), or have restricted programmability. They have a fixed value at power-on, depending mostly upon hard local controls.

power-on reset flags: Device flags to which a TRUE or FALSE value is assigned that determines the resetting of **device** functions at power-on (e.g., power-on-status-clear flag, see 10.25).

macros: Predefined program message units than can be executed by the **device** upon receipt of the macro label.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

device events.: Events that occur within or are detected by a **device** function. As distinct from conditions, which are identified as states, the changes of **device** states are considered as events. Events are generally reported only once (e.g., process state transitions, error detected, local control occured, power-on detect, etc.).

D.1.2 Device Functional Layer Definitions

Device functions: Consists of device-status functions and device-specific functions.

device-status functions: Functions that are identified in the IEEE 488.2 Status Reporting Model, see Section 11., and whose settings cannot be programmed by IEEE 488.2 common commands from the Status and Event group, see Table 10-2. Although the structure of IEEE 488.2 standard status functions is specified by this standard, they are part of layer D because they are controlled by device-specific program messages.

device-specific functions: Device-dependent functions that are neither defined nor identified by IEEE Std 488.1-1987 [4] or this standard.

IEEE 488.2 common functions: Device functions whose settings can be programmed or whose state can be queried by IEEE 488.2 defined common program messages. The IEEE 488.2 common functions are classified into groups that are defined in Table 10-1. IEEE 488.2 common functions consist of IEEE 488.2 standard status functions and IEEE 488.2 common operation functions.

IEEE 488.2 standard status functions: Functions identified in the IEEE 488.2 Status Reporting Model, see Section 11., whose settings can be programmed by IEEE 488.2 common commands from the Status and Event groups, see Table 10-1).

IEEE 488.2 common operation functions: IEEE 488.2 common functions that are neither IEEE 488.2 standard status functions nor belong to the IEEE 488.2 message exchange interface functions.

IEEE 488.2 message exchange interface functions: These are the functions that belong to the Message Exchange Interface, as defined in 6.1. The Input Buffer, Parser, Execution Control, Response Formatter, Output Queue, Trigger Control, and Message Exchange Control are part of the message exchange interface functions.

IEEE 488.1 Interface Functions: These are the I/O Control functions as defined in 6.1.4.

D.2 *RST Reset Summary

The common command *RST sets several functions to a specific default state. See 10.32 and Table D-1.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table D-1—*RST Reset Summary

DEVICE FUNCTIONAL CLASS	DEVICE FUNCTIONAL LAYER					
	Device Specific	**Device Status**	**Standard Status**	**Common Operation**	**Message Exchange Interface**	**IEEE 488.1 I.F.**
Configuration settings	–			–		–
Current **device** settings	+	–	–	+		
Device states	+	–		+	–	
Operation modes	+				–	
Memorized **device** settings	–			–		
Protected settings	–			–		
Power-on reset flags	–			–		
Macros	–			–		
Device events		–	–			–

\+ function affected by *RST
– function not affected by *RST
<no entry> not applicable

D.3 dcas Reset Summary

The Device Clear Active state requirements are summarized in 5.8 and Table D-2.

D.4 *CLS Reset Summary

The *CLS common command is described in 10.3, Section 11., and Table D-3.

D.5 Power-On Reset Summary

Power-on requirements are described in 5.12 and Table D-4.

D.6 Reset Summary

This section contains a summary of the reset requirements for all device functions.

In addition to the classification and layering, reset requirements are mentioned for each group. In Table D-5, the grouped functionality is not affected by *RST, *CLS, dcas, or pon when no requirements are mentioned.

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table D-2—dcas Reset Summary

DEVICE FUNCTIONAL CLASS	DEVICE FUNCTIONAL LAYER					
	Device Specific	**Device Status**	**Standard Status**	**Common Operation**	**Message Exchange Interface**	**IEEE 488.1 I.F.**
Configuration settings	–			–		–
Current **device** settings	+	–	–	–	–	
Device states	–	–		+	+	
Operation modes	–				+	
Memorized device settings	–			–		
Protected settings	–			–		
Power-on reset flags	–			–		
Macros	–			–		
Device events		–	–			–

\+ function affected by **dcas**
– function not affected by **dcas**
<no entry> not applicable

Table D-3—*CLS Reset Summary

DEVICE FUNCTIONAL CLASS	DEVICE FUNCTIONAL LAYER					
	Device Specific	**Device Status**	**Standard Status**	**Common Operation**	**Message Exchange Interface**	**IEEE 488.1 I.F.**
Configuration settings	–			–		–
Current **device** settings	–	–	–	–		
Device states	–	+		+	–	
Operation modes	–				–	
Memorized **device** settings	–			–		
Protected settings	–			–		
Power-on reset flags	–			–		
Macros	–			–		
Device events		+	+			–

\+ function affected by *CLS
– function not affected by *CLS
<no entry> not applicable

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table D-4—Power-On Reset Summary

DEVICE FUNCTIONAL CLASS	DEVICE FUNCTIONAL LAYER					
	Device Specific	**Device Status**	**Standard Status**	**Common Operation**	**Message Exchange Interface**	**IEEE 488.1 I.F.**
Configuration settings	–			–		–
Current device settings	+(1)	[posc]	posc	x		
Device states	x	x		+	+	
Operation modes	+				+	
Memorized device settings	x			x		
Protected settings	–			–		
Power-on reset flags	–			–		
Macros	plus;			+		
Device events		x	x			+

+	function affected by CLS
–	function not affected by CLS
<no entry>	not applicable
x	may or may not be affected by pon
posc	controlled by posc-flag value (when flag is implemented
[posc]	may or may not be controlled by posc-flag value (1) see 5.12

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Table D-5—Reset Summary for All Layers

DEVICE FUNCTIONAL CLASS	DEVICE FUNCTIONAL LAYER					
	Device Specific	**Device Status**	**Standard Status**	**Common Operation**	**Message Exchange Interface**	**IEEE 488.1 I.F.**
Configuration settings				*IDN? response *OPT? response Resource description		Bus address
Current device set- tings	*RST, pon (1)	Device Event Status Enable Registers ---------- [posc]	Service Request Enable Reg. Std. Event Status Enable Reg. [posc]	*DDT Trigger commands Enable/ Disable macros *PCB address ---------- *RST, [pon]		
Device states	*RST, [pon]	Device Queues *CLS, [pon]		OCC and OCQ functions ---------- RST, dcas , *CLS, pon	Parser, Execution Control, Response Formatter, Input Buffer, Output Queue dcas, pon	
Operation modes	*RST, pon				Automatic System Config. (*AAD/*DLF functions) ---------- dcas, pon	
Memorized device settings	[pon]			* SAV/*RCL registers ---------- [pon]		
Protected settings				*PUD data		
Power-on reset flags				*PSC (posc) flag		
Macros	[pon]			[pon], *PMC		
Device events		Device Event Registers Status Bite bits 0-3 and 7) ---------- *CLS	Standard Event Register Status Byte (bits 4 and 5) ist message ---------- *CLS			rsv message ---------- pon, *CLS

*RST	functions to be set to initial value on receipt of *RST
*CLS	functions to be cleared upon receipt of *CLS
dcas	functions cleared or reset upon receipt of dcas message
pon	functions to be set to initial value when power-on occurs
[pon]	functions may or may not be set to initial value when power-on occurs
posc	clearing or resetting of function at power-on depends upon the value of the power-on-status-clear flag, when implemented
[posc]	clearing or resetting of function at power-on may or may not depend upon the value of the power-on-status-clear flag
(1)	see 5.12

Published by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex E List of Participants (Informative)

At the time that this standard was completed, the working group had the following membership:

Bob Cram, *Chair*

L. Atchison	W. B. Groseclose	B. Nowlin
G. Blyth	C. Hagerling	R. Oblad
M. Brindle	M. G. Harrison	J. Pieper
R. Chojnacki	M. Hinds	A. Preuss
J. A. Colby	K. E. Johansen	L. Sollman
G. Cushing	S. G. Klaud	S. Tarr
P. D′Angelo	S. Lomas	D. Taylor
P. Davis	D. Lyons	A. Tracht
S. Greer	G. Meldrum	

At the time that it balloted and approved this standard for submission to the IEEE Standards Board, the balloting committee had the following membership:

L. Atchison	P. D′Angelo	W. B. Groseclose
S. E. Barryte	P. Davis	M. Harrison
D. Boyle	C. G. Gorss, Jr.	D. C. Hart
R. Cram	B. Gould	S. G. Klaud
G. Cushing	S. Greer	S. Lomas
D. Lyons	W. Nowlin	L. Sollman
M. Maciejewski	R. Oishi	P. Stuckert
A. Meloni	E. Sacher	C. Thomsen
R. H. Nesson	J. M. Schachner	A. Tracht

When the IEEE Standards Board approved this standard on June 18, 1992, it had the following membership:

Marco W. Migliaro, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Dennis Bodson	Donald N. Heirman	T. Don Michael *
Paul L. Borrill	Ben C. Johnson	John L. Rankine
Clyde Camp	Walter J. Karplus	Wallace S. Read
Donald C. Fleckenstein	Ivor N. Knight	Ronald H. Reimer
Jay Forster *	Joseph Koepfinger *	Gary S. Robinson
David F. Franklin	Irving Kolodny	Martin V. Schneider
Ramiro Garcia	D. N. “Jim” Logothetis	Terrance R. Whittemore
Thomas L. Hannan	Lawrence V. McCall	Donald W. Zipse

* Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal	Richard B. Engelman	Stanley Warshaw
James Beall	David E. Soffrin	

Adam Sicker
IEEE Standards Project Editor

--`,,`,,,`-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
Licensed by IEC under licence from IEEE. © 1992 IEEE. All rights reserved.
No reproduction or networking permitted without license from IHS Not for Resale

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Standards Survey

The IEC would like to offer you the best quality standards possible. To make sure that we continue to meet your needs, your feedback is essential. Would you please take a minute to answer the questions overleaf and fax them to us at +41 22 919 03 00 or mail them to the address below. Thank you!

Customer Service Centre (CSC)

International Electrotechnical Commission
3, rue de Varembé
1211 Genève 20
Switzerland

or

Fax to: **IEC**/CSC at +41 22 919 03 00

Thank you for your contribution to the standards-making process.

Nicht frankieren
Ne pas affranchir

Non affrancare
No stamp required

A Prioritaire

RÉPONSE PAYÉE

SUISSE

Customer Service Centre (CSC)
International Electrotechnical Commission
3, rue de Varembé
1211 GENEVA 20
Switzerland

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Q1 Please report on **ONE STANDARD** and **ONE STANDARD ONLY**. Enter the exact number of the standard: *(e.g. 60601-1-1)*

.....

Q2 Please tell us in what capacity(ies) you bought the standard *(tick all that apply).* I am the/a:

purchasing agent ❑
librarian ❑
researcher ❑
design engineer ❑
safety engineer ❑
testing engineer ❑
marketing specialist ❑
other.....

Q3 I work for/in/as a: *(tick all that apply)*

manufacturing ❑
consultant ❑
government ❑
test/certification facility ❑
public utility ❑
education ❑
military ❑
other.....

Q4 This standard will be used for: *(tick all that apply)*

general reference ❑
product research ❑
product design/development ❑
specifications ❑
tenders ❑
quality assessment ❑
certification ❑
technical documentation ❑
thesis ❑
manufacturing ❑
other.....

Q5 This standard meets my needs: *(tick one)*

not at all ❑
nearly ❑
fairly well ❑
exactly ❑

Q6 If you ticked NOT AT ALL in Question 5 the reason is: *(tick all that apply)*

standard is out of date ❑
standard is incomplete ❑
standard is too academic ❑
standard is too superficial ❑
title is misleading ❑
I made the wrong choice ❑
other .....

Q7 Please assess the standard in the following categories, using the numbers:
(1) unacceptable,
(2) below average,
(3) average,
(4) above average,
(5) exceptional,
(6) not applicable

timeliness.....
quality of writing.....
technical contents.....
logic of arrangement of contents .....
tables, charts, graphs, figures.....
other .....

Q8 I read/use the: *(tick one)*

French text only ❑
English text only ❑
both English and French texts ❑

Q9 Please share any comment on any aspect of the IEC that you would like us to know:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

--`,,`,,,-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

ISBN 2-8318-7441-6

CODE PRIX XL

For price, see current catalogue

ICS 17.220.20; 25.040.40; 35.200

Typeset and printed by the IEC Central Office
GENEVA, SWITZERLAND

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale