

IEEE Standard for Microprocessor Operating System Interfaces (MOSI)

Circuits and Devices

Communications Technology

Computer

Sponsored by the
Technical Committee on Microprocessors and Microcomputers
of the IEEE Computer Society

*Electromagnetics and
Radiation*

Energy and Power

Industrial Applications

*Signals and
Applications*

*Standards
Coordinating
Committees*

IEEE Std 855-1990



Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

October 29, 1990

SH13474

THIS PAGE WAS
BLANK IN THE ORIGINAL

IEEE Standard for Microprocessor Operating System Interfaces (MOSI)

Sponsor
**Technical Committee on
Microcomputers and Microprocessors
of the
IEEE Computer Society**

Approved May 31, 1990
IEEE Standards Board

Abstract: IEEE Std 855-1990, *IEEE Standard for Microprocessor Operating System Interfaces (MOSI)*, defines an interface between application programs and their operating environments. It is designed to be compatible with most operating systems in common use in microcomputer environments, within the scope of services that the various operating systems provide. It can be used to access the services of a variety of operating systems in both small-scale and large-scale environments.
Keywords: interface, microprocessor, operating environment, operating system.

Library of Congress Catalog Number 90-084889

ISBN 1-55937-037-8

Copyright © 1990 by

**The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA**

*No part of this publication may be reproduced in any form,
in an electronic retrieval system or otherwise,
without the prior written permission of the publisher.*

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE 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 IEEE 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 once 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.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

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 all 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 technical 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 on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Foreword

[This Foreword is not part of IEEE Std 855-1990, IEEE Standard for Microprocessor Operating System Interfaces (MOSI).]

This standard represents over ten years of effort by a number of dedicated persons, whose goal has been to develop a set of interface conventions to enhance the portability of application programs, in their access of operating system services, as they are transported across various operating environments.

The resulting standard for Microprocessor Operating System Interfaces, commonly referred to as the MOSI standard, defines the form, parameters, and operation of a collection of *functions* that provide access to commonly used operating system services.

The MOSI standard defines an *interface* between a program and its operating environment, while making limited assumptions about these entities themselves. *It is not an operating system standard.* Moreover, it does not define or assume a specific file system, programming language, or computer architecture.

The MOSI standard is not based on any single existing operating system. It is designed, however, to be compatible with most operating systems in common use in microcomputer environments, within the scope of services provided by the various operating systems. Preliminary implementation experience has shown MOSI to be a workable interface that can be used to access the services of a variety of operating systems in both small-scale and large-scale environments.

Application programs must access the MOSI interface using the facilities of their chosen programming language. The goal of portability requires that a common access mechanism, or binding, be established between each programming language and the MOSI interface. The definition of such language bindings falls outside the scope of the MOSI Working Group. However, to provide for the effective usability of the MOSI interface, a series of proposed bindings for common programming languages is included in the appendixes to this standard.

This standard was prepared by Working Group 855 of the Microprocessor Standards Committee, sponsored by the Technical Committee on Microprocessors and Microcomputers of the IEEE Computer Society. It is a substantial revision and extension of IEEE Std 855-1985, IEEE Trial-Use Standard Specifications for Microprocessor Operating Systems Interfaces. At the time the standard was approved, the active membership of the working group was as follows:

James D. Mooney, *Chair*

Steven Boylan
James Isaak

Takashi Katsukawa
Alan Marcum

Rajan Shastri
Robert G. Stewart

The following persons were on the balloting committee that approved this standard for submission to the IEEE Standards Board:

Mohammad Al-Malki
Duane L. Anderson
Ed Andert
Frank Artale
Michael Atkinson
Robert Baker
John Black
Winsor A. Brown
Todd Busniewski
Clyde Camp
Anthony Carrato
David Cohen
Paul Cook

James Davis
Su Dongzhuang
Andre Drew
Sourav Kumar Dutta
Ray Ellis
Wayne Fischer
Gordon Garmaise
Jack Goldstein
Sandor Halasz
Herbert Hecht
James Isaak
Richard M. Karcich
David Keeney

Hubert Kirmann
Ezzat Korany
Takahiko Kuki
Tom Kurihara
Glen Langdon
Min-Suk Lee
Wayne Loucks
W. Barry Mallett
William McDonald
Bruce Millard
Donald Miller
John E. Montague
James D. Mooney

Donald Mounday
Cuong Nguyen
Shlomo Pri-Tal
William Rinehuls
Hans Roosli
Norman Schneidewind
Paul Schofield

Donald Schricker
Jeffrey Shaw
Philip Shutt
Mike Smolin
Burton Strauss
Serge Szukalow
Darius Tanksalvala
Ahmed Tantawi

Robert Tripi
Yusuf Tuncel
David L. Umbaugh
Carl Warren
Fritz Whittington
Qiufeng Wu
Oren Yuen

When the IEEE Standards Board approved this standard on May 31, 1990, it had the following membership:

Marco W. Migliaro, *Chairman*

James M. Daly, *Vice Chairman*

Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Fletcher J. Buckley
Allen L. Clapp
Stephen R. Dillon
Donald C. Fleckenstein
Jay Forster*
Thomas L. Hannan

Kenneth D. Hendrix
John W. Horch
Joseph L. Koepfinger*
Irving Kolodny
Michael A. Lawler
Donald J. Loughry
John E. May, Jr.

Lawrence V. McCall
L. Bruce McClung
Donald T. Michael*
Stig Nilsson
Roy T. Oishi
Gary S. Robinson
Terrance R. Whittemore
Donald W. Zipse

*Member Emeritus

Contents

CHAPTER	PAGE
1. General	9
1.1 Scope	9
1.2 Name	9
1.3 Objective	9
1.4 Definitions	10
1.4.1 Standard Definitions	10
1.4.2 General System Terms	10
1.5 Capabilities	11
1.6 Future Revisions	12
2. Mechanics of the Standard	13
2.1 Format of a Standard Presentation	13
2.2 Definition of Parameter Types	13
2.3 Errors	14
2.3.1 Error Class Codes	14
2.3.2 Error Reason Codes	15
2.4 Exceptions	15
2.4.1 Exception Class Codes	15
2.4.2 Exception Subclass Codes	16
2.5 Time Units and Delays	16
2.6 Standard Functions	17
3. Memory Management	19
3.1 General Discussion	19
3.2 Module 1: Simple Memory Management	19
3.2.1 ALLOCATE_MEMORY	19
3.2.2 FREE_MEMORY	20
3.2.3 GET_ALLOCATION_UNIT	21
3.2.4 GET_AMOUNT_REMAINING	21
3.2.5 GET_BLOCK_SIZE	21
3.3 Module 2: Multispace Memory Management	22
3.3.1 MAKE_POOL	22
3.3.2 FREE_POOL	23
3.3.3 ALLOCATE_FROM_POOL	23
3.3.4 GET_GLOBAL_MEMORY_ID	24
3.3.5 GET_STORAGE_ID	25
4. Time Management	27
4.1 General Discussion	27
4.2 Module 1: Simple Delay	27
4.2.1 DELAY	27
4.3 Module 2: Timer Management	28
4.3.1 CONNECT_TIMER	28
4.3.2 DISCONNECT_TIMER	29
4.3.3 INITIALIZE_TIMER	29
4.3.4 CONTROL_TIMER	30
4.3.5 READ_TIMER	31
4.3.6 GET_EXPIRED_TIMER	32
4.4 Module 3: Time of Day	32
4.4.1 GET_TIME	32
4.4.2 GET_DATE	33



CHAPTER	PAGE
4.5 Module 4: Delay Until Time	34
4.5.1 SLEEP_UNTIL	34
5. Data Transfer	35
5.1 General Discussion	35
5.2 Module 1: Device I/O	36
5.2.1 CONNECT_DEVICE	36
5.2.2 DISCONNECT_DEVICE	37
5.2.3 READ_DEVICE_STATUS	37
5.2.4 WRITE_DEVICE_CONTROL	38
5.2.5 READ_DEVICE_DATA	39
5.2.6 WRITE_DEVICE_DATA	40
5.2.7 WAIT_DEVICE	41
5.2.8 GET_DEVICE_CHARACTERISTICS	42
5.3 Module 2: Synchronous File I/O	43
5.3.1 CONNECT_FILE	45
5.3.2 DISCONNECT_FILE	46
5.3.3 CREATE_FILE	46
5.3.4 DELETE_FILE	48
5.3.5 EXTEND_FILE	48
5.3.6 TRUNCATE_FILE	49
5.3.7 OPEN_FILE	50
5.3.8 CLOSE_FILE	51
5.3.9 READ_FILE	51
5.3.10 WRITE_FILE	53
5.3.11 SEEK_FILE	54
5.3.12 GET_FILE_INFORMATION	55
5.3.13 RENAME_FILE	57
5.3.14 GET_ACCESS_CONTROL	58
5.3.15 CHANGE_ACCESS_CONTROL	58
5.3.16 GET_USER_IDENTITY	59
5.4 Module 3: Asynchronous File I/O	60
5.4.1 INITIATE_READ	60
5.4.2 INITIATE_WRITE	61
5.4.3 INITIATE_SEEK	62
5.4.4 WAIT_TRANSFER	63
5.4.5 ABORT_TRANSFER	65
5.5 Module 4: Multiple Directories	66
5.5.1 GET_WORKING_DIRECTORY	66
5.5.2 CHANGE_WORKING_DIRECTORY	66
5.5.3 CREATE_DIRECTORY	67
5.5.4 DELETE_DIRECTORY	67
5.5.5 GET_DIRECTORY_ENTRY	68
6. Data Management	71
6.1 General Discussion	71
6.2 Module 1: File Locking	71
6.2.1 LOCK_FILE	72
6.2.2 LOCK_RECORDS	73
6.2.3 UNLOCK_FILE	74
6.2.4 UNLOCK_RECORDS	75
6.2.5 CHANGE_LOCK	76
6.3 Module 2: File Replication	77

CHAPTER	PAGE
6.3.1 REPLICATE_FILE	77
6.4 Module 3: Atomic Transactions	78
6.4.1 BEGIN_TRANSACTION	79
6.4.2 BIND_TO_TRANSACTION	79
6.4.3 COMMIT_TRANSACTION	80
6.4.4 ABORT_TRANSACTION	81
7. Process Management	83
7.1 General Discussion	83
7.2 Module 1: Single Process	84
7.2.1 INITIALIZE_PROGRAM	84
7.2.2 TERMINATE_PROGRAM	85
7.3 Module 2: Multiple Processes	85
7.3.1 CREATE_PROCESS	85
7.3.2 DESTROY_PROCESS	87
7.3.3 GET_PROCESS_STATUS	87
7.3.4 GET_PROCESS_INFO	88
7.3.5 CHANGE_PRIORITY	89
7.4 Module 3: Suspend and Resume	89
7.4.1 SUSPEND_PROCESS	90
7.4.2 RESUME_PROCESS	90
8. Process Synchronization and Communcation	92
8.1 General Discussion	92
8.2 Module 1: Semaphores	93
8.2.1 ALLOCATE_SEMAPHORE	93
8.2.2 SIGNAL_SEMAPHORE	94
8.2.3 WAIT_SEMAPHORE	95
8.2.4 DEALLOCATE_SEMAPHORE	96
8.3 Module 2: Multiple Semaphores	96
8.3.1 MULTIPLE_SIGNAL_SEMAPHORE	97
8.3.2 MULTIPLE_WAIT_SEMAPHORE	97
8.4 Module 3: Messages	98
8.4.1 ALLOCATE_MAILBOX	98
8.4.2 SEND_MESSAGE	99
8.4.3 RECEIVE_MESSAGE	100
8.4.4 DEALLOCATE_MAILBOX	101
8.5 Module 4: Messages With Responses	102
8.5.1 RESPOND_MESSAGES	102
8.5.2 WAIT_RESPONSE	103
8.6 Module 5: Event Flags	104
8.6.1 ALLOCATE_EVENT_FLAG	104
8.6.2 DEALLOCATE_EVENT_FLAG	105
8.6.3 SET_EVENT_FLAG	105
8.6.4 READ_EVENT_FLAG	106
8.6.5 WAIT_EVENT_FLAG	106
9. Interface With the Environment	108
9.1 General Discussion	108
9.2 Module 1: System Characteristics	108
9.2.1 GET_SYSTEM_CHARACTERISTICS	108
9.3 Module 2: Command String Acquisition	109
9.3.1 GET_COMMAND_STRING	109

CHAPTER	PAGE
9.3.2 GET_TOKEN	110
9.4 Module 3: Direct Argument Acquisition	111
9.4.1 GET_ARGUMENT_COUNT	111
9.4.2 GET_ARGUMENT	111
9.5 Module 4: Overlay Management	112
9.5.1 LOAD_OVERLAY	112
9.6 Module 5: I/O Assignments	113
9.6.1 GET_IO_ASSIGNMENTS	113
9.6.2 CHANGE_IO_ASSIGNMENTS	114
10. Exception Handling	116
10.1 General Discussion	116
10.2 Module 1: Default Handler	117
10.2.1 DISABLE_EXCEPTION_CLASS	117
10.2.2 ENABLE_EXCEPTION_CLASS	118
10.2.3 GET_ENABLE_STATUS	119
10.3 Module 2: Single Exception Handler	119
10.3.1 GET_EXCEPTION_HANDLER	119
10.3.2 SET_EXCEPTION_HANDLER	120
10.3.3 RESET_DEFAULT_HANDLER	120
10.3.4 RAISE_EXCEPTION	120
10.3.5 GET_EXCEPTION_CODE	121
10.3.6 EXIT_FROM_HANDLER	122
10.4 Module 3: Exception Class Handlers	122
10.4.1 GET_EXCEPTION_CLASS_HANDLER	122
10.4.2 SET_EXCEPTION_CLASS_HANDLER	123
11. Compliance With the Standard	124
11.1 General Discussion	124
11.2 Recommended Configurations	124
11.2.1 Realtime Support, Level 1	125
11.2.2 Realtime Support, Level 2	126
11.2.3 Realtime Support, Level 3	127
11.2.4 Development Support, Level 1	127
11.2.5 Development Support, Level 2	128
11.2.6 Commercial Support, Level 1	129
11.2.7 Complete Support	129
APPENDIXES	
Appendix A Error Summary	130
Appendix B Exception Code Summary	134
Appendix C Future Extensions	135
Appendix D Pascal Language Binding	136
Appendix E C Language Binding	157
Appendix F FORTRAN Language Binding	183
Appendix G COBOL Language Binding	197
Appendix H Ada Language Binding	206
Appendix I PL/M Language Binding	228

Chapter 1

General

1.1 Scope

This standard applies to program interfaces for microprocessor operating systems. These program interfaces are used by microprocessor applications to interface with operating system services. The standard is not an operating system standard; rather, it defines an interface between an operating system and an application program. This interface can be implemented directly by an operating system, by a separate software layer or procedure library, or by translation of the program prior to execution.

1.2 Name

The title of this standard is “IEEE Standard for Microprocessor Operating System Interfaces.” It is referred to as “MOSI” throughout the document.

1.3 Objective

The goal of MOSI is to facilitate the writing of portable application programs or systems. Using this standard, a programmer should be able to write programs that, with few or (ideally) no changes at the source level, may be run on many different *computer systems*, that is, particular machines running particular operating systems. This document specifies capabilities intended to support various microprocessor-based applications.

The computer hardware, operating system, and associated software that provide support for an application are referred to collectively as the *environment*. Three components interact in preparing and executing a program in most environments. The program is first processed by a *language translator*. The generated object module then runs under a particular *operating system* that controls a specific *physical machine*. The language, the operating system, and the machine architecture are all predetermined and highly variable. MOSI makes no attempt to standardize or constrain the structure of any of these. However, the standard provides the application programmer with a dependable mechanism to be used to communicate within a chosen language with the operating system. It is the responsibility of a conforming operating system or language translator to provide the appropriate interfaces specified by the standard.

The process of transporting an application to a new environment consists of obtaining a copy of the source program, modifying it if necessary, processing it with a translator that targets the desired system, and executing the code produced by the translator. If no change to the source is required, there may be perfect portability. Minor changes to the source are usually an acceptable compromise. This standard is designed to enhance that portability.

To be widely useful, this standard must be applicable to a variety of microprocessor operating systems and application languages, including those already in widespread use. To meet this objective, the standard

- Makes few assumptions about the machine architecture.
- Makes few assumptions about the structure or capabilities of the operating system, including file system, user interface, process management, etc.
- Specifies general data types usable within the context of common programming languages.

1.4 Definitions

1.4.1 Standard Definitions

function. A primitive operation on system-controlled resources. This standard defines a collection of functions together with suitable input and output parameters.

capability. A set of functions that are logically related by the common set of resources on which they operate.

capability module. A set of functions within a capability that provide a class of support. Each module is intended to be provided in its entirety.

configuration. A collection of capability modules.

1.4.2 General System Terms

asynchronous. Describes an activity initiated by a function that is not necessarily complete when the function returns.

bit. A contraction of the term “binary digit”; a unit of information represented by either a zero or a one.

block. A group of data that is contiguous in nature.

byte. A binary bit string operated on as a unit and usually eight bits long and capable of holding one character in the local character set.

event flag. A single bit variable used to represent the occurrence of a particular event.

file. A set of related records usually treated as a named unit of storage.

interface. A shared boundary between two layers or modules of software.

process. A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources.

program. A collection of processes working together to accomplish a common task.

record. A collection of related data or words treated as a unit and saved in a position-dependent fashion within a file of other such units.

seek. An activity that positions a pointer at a specific location within a data file.

semaphore. A system variable used to synchronize concurrent processes by indicating whether an action has been completed or an event has occurred.

synchronous. Describes an activity specified by a function that is expected to be complete when the function returns.

word. An ordered set of bytes or bits that is the normal unit in which information may be stored, transmitted, or operated on within a given computer.

1.5 Capabilities

The functions defined by this standard are grouped according to their purpose in a number of categories. Each category is called a capability and includes one or more modules. Each module in a capability is comprised of a set of functions that are logically related by the common set of resources on which they operate. The collection of capabilities defined by the standard is intended to encompass all those generic capabilities that are recurrent throughout various environments. This collection is listed below.

- Memory Management
- Time Management
- Data Transfer (Input-Output)
- Data Management
- Process Management
- Process Synchronization and Communication
- Interface with the Environment
- Exception Handling

These capabilities are described in detail in Chapters 3 through 10.

The operating system services required by an application for its operating environment can be stated unambiguously using this approach.

A number of sets of logically related functions occur commonly in various applications. To support such sets, MOSI defines several *recommended configurations*. Chapter 11 identifies each of these and describes their normal usage.

1.6 Future Revisions

This standard specifies a set of functions commonly available in operating systems and useful in portable application programs. The appropriate set of functions is continuing to evolve, and additional areas may be suitable candidates for standardization in the future. It is anticipated that future versions of this standard may be developed to address additional categories of service. Some candidate categories are listed in Appendix C.

Chapter 2 Mechanics of the Standard

Each definition in this standard consists of a formal specification and a prose description. Language bindings for several programming languages are included as examples in appendixes.

2.1 Format of a Standard Presentation

Name: FUNCTION_NAME (ip1, ip2, ...ipn), ({opr}, op1, ..., error).

ip1, ip2, etc., represent input items (if any), and opr, op1, etc., represent returned items. opr represents the one parameter, if any, that may be returned by the function call, for those languages supporting such a mechanism. It is denoted by braces { } and shall be listed first in the returned items. The error return is always listed last.

Usage: Prose description of the actions and effects of the function.

Input Items: A list of each item name, its type, and its purpose, including domain specifications.

Returned Items: A list of each returned item, its type, and its meaning, including range specifications.

Side Effects: Either the word “none” or a discussion of any side effects.

Error Conditions: Either the word “none” or a listing of each possible error and its meanings. Note that any function may return an error with reason code -1 (see 2.3.2), and, to eliminate repetition, this error return is not shown on each function.

2.2 Definition of Parameter Types

Each function parameter (input item or returned item) is defined to have one of the following data types:

systemtag. An identifier returned by one function for use by another. Its contents may not be examined or changed. Its form is implementation-dependent. A systemtag is valid systemwide (i.e., it is global) and may be passed between processes.

processtag. An identifier returned by one function for use by another. Its contents may not be examined or changed. Its form is system-dependent. A processtag is only valid within a given process and should not be passed between processes.

flag. A logical value that may be either true or false.

integer. A whole number that may be positive, negative, or zero. The range of an integer is at least $-32\,767$ to $+32\,767$.

long integer. A whole number that may be positive, negative, or zero. The range of a long integer is at least $(-2^{31} + 1)$ to $(+2^{31} - 1)$.

storageid. An identifier for a block of data. The identifier is not guaranteed to be valid outside the allocating process and should not be passed between processes.

string. A sequence of characters together with an indication of its length. In general, the standard expects ASCII characters. However, the standard does not preclude other character sets.

Systemtags and processtags representing dissimilar types of objects are not guaranteed to be distinct. The types systemtag, processtag, storageid, and string are assumed to have a distinguishable null value.

2.3 Errors

An error code shall be the last returned item for all functions and shall be of type storageid. The error code identifies a block of data containing four integers. On return from the function, the error code shall contain the following:

Integer	Contents
1	Either zero (no error) or the class number of the error.
2	Either zero (no error) or the reason number of the error.
3–4	System-dependent information. Not necessarily zero if there was no error.

In addition, for some functions, the error information may be encoded in returned values. A class number is a logical grouping of error codes as specified below. A reason number is a specific error within a class number.

Conformance to this standard guarantees to the user that if no error is returned by a MOSI function, the function was performed as requested. Also, if an error code indicating an invalid input item is returned, the function has not been attempted.

2.3.1 Error Class Codes

The following error class codes are defined by this standard:

Error Class Code	Error Class Description
0	No errors occurred
1	Memory management
2	Time management and scheduling
3	Data transfer
4	Data management
5	Process management
6	Process synchronization and communication
7	Interface with the environment
8	Exception handling
9-199	Reserved for future standardization
200-255	User-defined

2.3.2 Error Reason Codes

The error reason codes are independent among error classes with two exceptions: codes 0 and -1. Error reason code 0 (zero) has been uniformly defined to mean no error. Error reason code -1 has been uniformly defined to have the meaning "FUNCTION NOT IMPLEMENTED" for all error classes.

The error codes that may be returned by each function are listed and explained together with the function definition. All error codes defined in this standard are also summarized in Appendix A.

2.4 Exceptions

Exceptions define an interruption of the normal flow of an application in response to specific events. Examples of such events include attempts to divide by zero, depression of a "break" key, or hardware interrupts. Exceptions may be handled by either default or user-defined exception handlers. An exception class code and exception subclass code may be passed to the exception handler to identify the cause of the exception.

2.4.1 Exception Class Codes

The following table lists the exception class codes that are defined by this standard:

Exception Class Code	Exception Class Description
1	Procedural exception
2	Operator intervention
3	Hardware interrupt
4-99	Reserved for future standardization
100-255	User-defined

2.4.2 Exception Subclass Codes

The exception subclass codes are unique within each exception class. The subclass codes recognized by this standard are shown in the following table:

Class	Subclass	Exception
1	1	Divide by zero
1	2	Bounds error
1	3	Overflow
1	4	Underflow
1	5	Stack overflow
2	1	Break
3	1	Timer expired
3	2	Math coprocessor

This list is not exhaustive and may not represent those subclass codes supported by a specific implementation.

2.5 Time Units and Delays

A number of functions throughout this standard specify a time interval in two parts: an integer value, followed by a code designating the time units in which the value is measured. Time unit codes specify either a fixed time interval or an implementation-dependent interval called a *tick*. The value of a tick for a specific implementation may be obtained from the GET_SYSTEM_CHARACTERISTICS function (see 9.2.1). The time unit codes defined by this standard are as follows:

Time_Unit Code	Unit
-1	ticks
0	microseconds
1	milliseconds
2	seconds
3	minutes
4	hours

The accompanying time value specifies the number of units of the indicated type in the time interval. A value of zero indicates an interval of zero length. A value of -1 is defined to represent an unlimited time interval.

2.6 Standard Functions

The functions that are defined in Chapters 3 through 10 of this standard are considered MOSI *standard functions*. A complete list may be found in the Contents.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Chapter 3 Memory Management

3.1 General Discussion

The Memory Management capability includes functions that provide the interface to the memory management facility in a microcomputer operating system. Memory management is the dynamic acquisition and release of allocatable, directly addressable memory. Allocatable memory consists of all memory that is made available to the program by the memory management system.

The memory space in simple memory management consists of a single memory pool. Multispace memory management provides for allocation from pools that have specific characteristics.

Module 1 Simple Memory Management	Module 2 Multispace Memory Management
ALLOCATE_MEMORY	MAKE_POOL
FREE_MEMORY	FREE_POOL
GET_ALLOCATION_UNIT	ALLOCATE_FROM_POOL
GET_AMOUNT_REMAINING	GET_GLOBAL_MEMORY_ID
GET_BLOCK_SIZE	GET_STORAGE_ID

3.2 Module 1: Simple Memory Management

This module provides support for systems that provide dynamic allocation and deallocation of memory. All memory is logically viewed as a single memory pool. The allocation method is left to the discretion of the operating system.

3.2.1 ALLOCATE_MEMORY

Name: ALLOCATE_MEMORY (size), ({location}, error)

Usage: Obtains a specified amount of memory for use by the calling process. The amount allocated is at least the smallest integer multiple of the unit size (see 3.2.3) that is equal to or greater than the amount requested.

Input Items:

size. Non-negative long integer indicating the size, in bytes, of the desired block of memory. The value 0 shall indicate a request for the largest memory block currently available.

Returned Items:

location. Storageid of the allocated memory. In case of an error condition, the value returned has a null value.

Side Effects: The size of the free storage pool is decreased and the storage allocated becomes available for use by the caller.

Error Conditions:

Class	Reason	Error
1	1	INVALID MEMORY REQUEST — The requested size was meaningless or not supported.
1	2	INSUFFICIENT MEMORY — Enough contiguous memory was not available to support the request.

3.2.2 FREE_MEMORY

Name: FREE_MEMORY (location), (error)

Usage: Returns a previously allocated block of memory to the free storage pool, removing it from the memory available to the calling program. Further attempts at using this freed storage may result in errors, or may produce unexpected results since the memory referenced may have been reallocated to another process. The storageid is no longer valid after this call. Results of additional calls to FREE a previously freed block are not defined by this standard.

Input Items:

location. Storageid of the memory to be released.

Returned Items: None.

Side Effects: The amount of storage available for allocation increases.

Error Conditions:

Class	Reason	Error
1	3	INVALID MEMORY BLOCK — The <i>location</i> input item did not specify a valid storage block.

3.2.3 GET_ALLOCATION_UNIT

Name: GET_ALLOCATION_UNIT (), ((*unit_size*), error)

Usage: Returns the size, in bytes, of the smallest possible unit of allocation.

Input Items: None.

Returned Items:

unit_size. Integer whose value is the size in bytes of the unit of storage allocation. This is used implicitly by the allocation functions to calculate, in conjunction with the requested number of bytes, the actual number of bytes to be allocated.

Side Effects: None.

Error Conditions: None.

3.2.4 GET_AMOUNT_REMAINING

Name: GET_AMOUNT_REMAINING (), ((*total*), largest, error)

Usage: Tells the caller the amount of storage remaining unallocated that is available to that process. It is not guaranteed that this amount will be available at the time of a subsequent ALLOCATE request.

Input Items: None.

Returned Items:

total. Non-negative long integer whose value, in bytes, is the total amount of storage still unallocated.

largest. Non-negative long integer whose value is the number of bytes in the largest contiguous space still unallocated.

Side Effects: None.

Error Conditions: None.

3.2.5 GET_BLOCK_SIZE

Name: GET_BLOCK_SIZE (*location*), ((*size*), error)

Usage: Returns the size of the allocated storage block in bytes.

Input Items:

location. Storageid of a previously allocated storage block.

Returned Items:

size. Non-negative long integer whose value is the number of bytes in the specified storage block.

Side Effects: None.

Error Conditions:

Class	Reason	Error
1	3	INVALID MEMORY BLOCK — The <i>location</i> input item did not specify a valid storage block.

3.3 Module 2: Multispace Memory Management

This module provides for management of memory pools distinguished by various attributes, including shareability, volatility, speed, segmentation, or logical distinctions based on contents. Allocation of storage blocks from specific pools is supported, and a mechanism for memory sharing is included.

Distinctions among different types of memory are made using the *attributes* parameter of the MAKE_POOL function. Some attributes are defined by this standard; others are available to the implementation; yet others are left as user-defined (for implementing segmented memories, for example).

3.3.1 MAKE_POOL

Name: MAKE_POOL (size, attributes, pool_number, parameters) ({*pool_id*}, amount, error)

Usage: Creates a memory pool with the desired attributes. At least the specified number of bytes will be in the pool, unless that amount of memory is unavailable with the specified attributes. In this case, a smaller pool may be created.

Input Items:

size. Positive long integer specifying the number of bytes desired in the pool. A value of 0 requests the largest pool available with the specified attributes.

attributes. An integer describing the type of pool desired. Each attribute defined by the standard is assigned a value that is a power of two. Attributes may be combined by adding them. The defined attributes are as follows:

0	no special attributes
1	shareable pool
2	nonvolatile pool
4	guaranteed main-memory resident
8	fixed allocation unit (parameter then contains a long integer specifying the block size)
16–128	reserved for future standardization
256–32 768	implementation-defined

pool_number. Long integer containing a user-specified pool identification number. The pool numbered 0 is the default pool; no other special meaning may be assigned to pool number 0.

parameters. Storageid designating a block of memory containing additional information to describe the type of pool desired. Its content depends on the *attributes* value.

Returned Items:

pool_id. Systemtag denoting the pool.

amount. Non-negative long integer containing the actual size of the storage pool created. If this value is less than the requested size, an error will be returned as well.

Error Conditions:

Class	Reason	Error
1	1	INVALID MEMORY REQUEST — The requested size was meaningless or not supported.
1	2	INSUFFICIENT MEMORY — Enough memory with the specified attributes was not available.
1	4	NO SUCH POOL — No pool with the specified attributes can be constructed.

3.3.2 FREE_POOL

Name: FREE_POOL (pool_id), (error)

Usage: Frees all the storage associated with a storage pool.

Input Items:

pool_id. Systemtag, returned by MAKE_POOL, denoting the pool to be freed.

Returned Items: None.

Error Conditions:

Class	Reason	Error
1	5	INVALID POOL — The specified pool identifier did not denote a valid memory pool.
1	6	POOL ALREADY FREED — The specified pool has already been freed.

3.3.3 ALLOCATE_FROM_POOL

Name: ALLOCATE_FROM_POOL (pool_id, size), ({location}, error)

Usage: Allocates storage from a specified pool.

Input Items:

pool_id. Systemtag, obtained from MAKE_POOL, denoting the pool from which allocation should occur.

size. Non-negative long integer indicating the number of bytes desired. A value of 0 requests the largest amount of memory available in the specified pool.

Returned Items:

location. Storageid of the allocated memory.

Error Conditions:

Class	Reason	Error
1	1	INVALID MEMORY REQUEST — The requested size was meaningless or not supported.
1	2	INSUFFICIENT MEMORY — Enough memory was not available in the specified pool.
1	5	INVALID POOL — The specified pool identifier did not denote a valid memory pool.

3.3.4 GET_GLOBAL_MEMORY_ID

Name: GET_GLOBAL_MEMORY_ID (location), ({global_id}, error)

Usage: Translates a process-local storageid into a global systemtag that can be passed to other processes to allow memory sharing.

Input Items:

location. The storageid to be translated.

Returned Items:

global_id. A systemtag that, when translated using GET_STORAGE_ID, can be used to reference the same memory referenced by *location*. Note that this item does not necessarily permit access to the memory specified unless translated using GET_STORAGE_ID.

Error Conditions:

Class	Reason	Error
1	7	INVALID LOCATION — The specified <i>location</i> was not valid for this process.

- 1 8 LOCATION NOT SHAREABLE — The specified *location* cannot be made global.

3.3.5 GET_STORAGE_ID

Name: GET_STORAGE_ID (*global_id*) (*location*), error)

Usage: Translates a global memory id obtained from GET_GLOBAL_MEMORY_ID into a storageid usable for referencing memory directly.

Input Items:

global_id. A systemtag, obtained from GET_GLOBAL_MEMORY_ID, that references the desired memory.

Returned Items:

location. A storageid that will refer to the same memory as that referenced by *global_id*.

Error Conditions:

Class	Reason	Error
1	9	INVALID GLOBAL ID — The specified global memory identifier does not describe a valid shareable memory block.
1	10	GLOBAL MEMORY NOT ACCESSIBLE — The block specified by the global memory identifier cannot be accessed by this process.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Chapter 4 Time Management

4.1 General Discussion

This capability is intended to provide a standardized method for application programs to make use of system time and timers. Four capability modules have been defined. Module 1, Simple Delay, allows a process to delay itself for a specified period of real time. Module 2, Timer Management, supports the use and allocation of timers. Module 3, Time of Day, provides functions for obtaining the current date and time. Module 4, Delay Until Time, allows a process to suspend itself until a specific date and time.

Module 1 Simple Delay	Module 2 Timer Management	Module 3 Time of Day	Module 4 Delay Until Time
DELAY	CONNECT_TIMER	GET_TIME	SLEEP_UNTIL
	DISCONNECT_TIMER	GET_DATE	
	INITIALIZE_TIMER		
	CONTROL_TIMER		
	READ_TIMER		
	GET_EXPIRED_TIMER		

4.2 Module 1: Simple Delay

This module allows a process to delay itself for a specified period of real time.

4.2.1 DELAY

Name: DELAY (time_value, time_unit), (error)

Usage: Delays the calling process for a specified period of real time. The accuracy of the delay obtained is implementation-dependent.

Input Items:

time_value. Non-negative integer specifying the time value for the delay, as described in 2.5.

time_unit. Integer code that indicates the units in which the time value is expressed, as described in 2.5.

Returned Items: None.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	2	INVALID TIME VALUE — The time value parameter was invalid.
2	4	DELAY ABORTED — The delay did not occur or was for less than the requested time.
2	5	INVALID TIME UNIT — The selected time unit granularity was not supported.

4.3 Module 2: Timer Management

The timer management module allows the user to associate an event with the timeout of a timer. Upon expiration of a timer, the operating system raises an exception (see Chapter 10) for the process that owns the timer. The exception code will be class 3, subclass 1 to indicate “timer expired.” The identity of the timer that caused the exception may be found with the GET_EXPIRED_TIMER function.

The user of the timer is responsible for setting the exception handler before starting the timer. The accuracy of the timer is implementation-dependent.

4.3.1 CONNECT_TIMER

Name: CONNECT_TIMER (), ((timer_id), error)

Usage: Requests that a specific, private timer be allocated to the current process. The number of distinct timers a single process may allocate is implementation-dependent. The timer initially contains the count value of zero and is inactive (stopped).

Input Items: None.

Returned Items:

timer_id. Systemtag identifying the timer allocated.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	3	TIMER NOT AVAILABLE — No timer was available for allocation to the calling process.

4.3.2 DISCONNECT_TIMER

Name: DISCONNECT_TIMER (timer_id), (error)

Usage: Releases a timer previously allocated by CONNECT_TIMER.

Input Items:

timer_id. Systemtag, previously assigned by CONNECT_TIMER, identifying the timer to be disconnected.

Returned Items: None.

Side Effects: The timer is deallocated and is no longer associated with the calling process.

Error Conditions:

Class	Reason	Error
2	1	INVALID TIMER ID — The timer was not created by this process or was unknown to this process.

4.3.3 INITIALIZE_TIMER

Name: INITIALIZE_TIMER (timer_id, time_value, time_unit, timer_mode), (error)

Usage: Assigns an initial count value and mode of operation to a timer. The timer must have been allocated using CONNECT_TIMER. The timer shall be stopped when this function is invoked. If the timer is running, an error is signaled and the result is unpredictable. This function does not start the timer.

Input Items:

timer_id. Systemtag, previously assigned by CONNECT_TIMER, identifying the timer to be initialized.

time_value. Non-negative integer specifying the time value for the delay, as described in 2.5.

time_unit. Integer code that indicates the desired units in which the time value is expressed, as described in 2.5.

timer_mode. Integer code that specifies the mode of operation of the timer. Modes recognized by this standard are listed in the table below. See the CONTROL_TIMER function (4.3.4) for information on how these modes function.

Value	Mode
1	One shot
2	Continuous

Returned Items: None.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	1	INVALID TIMER ID — The timer was not created by this process or was unknown to this process.
2	2	INVALID TIME VALUE — The time value parameter was invalid.
2	5	INVALID TIME UNIT — The selected time unit granularity was not supported.
2	9	INVALID MODE — The specified mode was invalid.
2	13	TIMER NOT STOPPED — The timer was not stopped when the function was invoked.

4.3.4 CONTROL_TIMER

Name: CONTROL_TIMER (timer_id, timer_control), (error)

Usage: Starts, stops, or resets a timer. The *timer_control* parameter specifies the operation to be performed. The timer counts downward to zero in the units of time specified in INITIALIZE_TIMER. It generates an exception (see Chapter 10) when a zero value is reached. The exception code is class 3, subclass 1. The timer identifier may be obtained with the GET_EXPIRED_TIMER function.

Two operating modes are supported: *one shot* and *continuous*. The mode is selected with the INITIALIZE_TIMER function. In the one-shot mode, the timer counts down to zero, issues the exception, and continues counting downward (“rolls over”). It will not issue further exceptions, however, until it has first been reset. This rollover is provided for applications where it may be important to know the total elapsed time when the exception is being acted on.

In the continuous mode, the timer counts down to zero, issues the exception, resets itself to the initial count, and continues counting downward. It will continue issuing exceptions at periodic intervals until stopped.

If the initialized value is zero, the timer will only issue an exception once whenever it is started after being reset.

Stopping or starting the timer does not clear or reset it. Resetting the timer stops it if it is running.

Input Items:

timer_id. Systemtag, previously assigned by CONNECT_TIMER, identifying the timer to be controlled.

timer_control. Integer code that controls the operation to be performed on the timer as follows:

Value	Action
0	Stop timer
1	Reset timer
2	Start timer
3	Reset and Start timer

Returned Items: None.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	1	INVALID TIMER ID — The timer was not created by this process or was unknown to this process.
2	8	INVALID CONTROL CODE — The timer control code specified was invalid.

4.3.5 READ_TIMER

Name: READ_TIMER (timer_id), ((time_value), time_unit, error)

Usage: Determines the current time value of a running or stopped timer allocated to the process.

Input Items:

timer_id. Systemtag, previously assigned by CONNECT_TIMER, identifying the timer to be read.

Returned Items:

time_value. Integer giving the current time value for the timer, as described in 2.5.

time_unit. Integer code indicating the units in which the time value is expressed, as described in 2.5. This will be the same time unit code specified when the INITIALIZE_TIMER function was invoked.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	1	INVALID TIMER ID — The timer was not created by this process or was unknown to this process.

4.3.6 GET_EXPIRED_TIMER

Name: GET_EXPIRED_TIMER (), ((timer_id), error)

Usage: Determines the most recently expired timer that caused an exception. If more than one timer has expired, only the identity of the most recent one can be obtained.

Input Items: None.

Returned Items:

timer_id. Systemtag, previously assigned by CONNECT_TIMER, identifying the timer that most recently expired. This value will be null if no timer expired.

Side Effects: Once an expired timer has been queried by this function, its indication is reset so that further calls will not identify the same timer unless it causes another exception.

Error Conditions:

Class	Reason	Error
2	7	NO EXPIRED TIMER — There is no timer that has expired and not yet been queried by GET_EXPIRED_TIMER.

4.4 Module 3: Time of Day

This module provides functions that allow applications to determine the current date and time. It assumes the system has a system calendar and time-of-day clock that is available to the user. The time and date values are provided in a standard encoding so that the user can provide formatting routines as desired.

4.4.1 GET_TIME

Name: GET_TIME (), ((time_of_day), error)

Usage: Obtains the current time-of-day. The accuracy of the value obtained is implementation-dependent. The time value is returned as a long integer, and may be converted to a text string in any desired format by separate procedures that are not specified by this standard.

Input Items: None.

Returned Items:

time_of_day. Long integer representing the number of milliseconds since the beginning of the current day. The value -1 indicates that the current time is not available.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	6	TIME OF DAY NOT SUPPORTED — The system does not support time-of-day.

4.4.2 GET_DATE

Name: GET_DATE (), ((date), time_of_day, error)

Usage: Obtains the current date and time-of-day. Provides a transformation between the operating system day encoding and the encoding specified by this standard. The date is defined as the number of days since January 1, 1980. This function also returns the time-of-day (as in GET_TIME) so two distinct calls do not have to be made, eliminating a possible one-day error around midnight.

Input Items: None.

Returned Items:

date. Integer representing the number of days since January 1, 1980. A value of -1 indicates that the current date is not available.

time_of_day. Long integer representing the number of milliseconds since the beginning of the current day. A value of -1 indicates that the current time is not available.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	11	DATE NOT SUPPORTED — The system does not maintain the current date.

2 12 **DATE AND TIME NOT SUPPORTED** — Neither the current date nor the current time are available from the operating system.

4.5 Module 4: Delay Until Time

4.5.1 SLEEP_UNTIL

Name: SLEEP_UNTIL (date, time_of_day), (error)

Usage: Delays the calling process until the specified date and time. If a date is specified that is before the current date, there is no delay and the **INVALID DATE** error is returned. If the date specified is the current date, and a time of day is specified that is before the current time, there is no delay, and no error is signaled.

Input Items:

date. Integer representing the date on which the calling process is to be awakened. The date is specified as the number of days since January 1, 1980. A value of -1 may be used to represent the current date.

time_of_day. Non-negative long integer representing the time in milliseconds on the specified day that the calling process is to be awakened.

Returned Items: None.

Side Effects: None.

Error Conditions:

Class	Reason	Error
2	14	INVALID TIME OF DAY — The time-of-day parameter was invalid.
2	4	DELAY ABORTED — The delay did not occur or was for less than the requested time.
2	6	TIME OF DAY NOT SUPPORTED — The system does not maintain the current date and time.
2	10	INVALID DATE — The date value specified was either before the current day or invalid.

Chapter 5 Data Transfer

5.1 General Discussion

This capability addresses the transfer of data among application storage areas (such as files), data sources and sinks (such as I/O devices), and main storage.

Four capability modules are defined. Module 1, Device I/O, covers direct interaction with a device. Module 2 defines functions for synchronous file I/O, and Module 3 deals with asynchronous file I/O. These modules support interaction with a named data space or file system. Finally, Module 4 supports the manipulation of multiple file directories.

Module 1 Device I/O	Module 2 Synchronous File I/O
CONNECT_DEVICE	CONNECT_FILE
DISCONNECT_DEVICE	DISCONNECT_FILE
READ_DEVICE_STATUS	CREATE_FILE
WRITE_DEVICE_CONTROL	EXTEND_FILE
READ_DEVICE_DATA	TRUNCATE_FILE
WRITE_DEVICE_DATA	OPEN_FILE
WAIT_DEVICE	CLOSE_FILE
GET_DEVICE_CHARACTERISTICS	READ_FILE
	WRITE_FILE
	SEEK_FILE
	GET_FILE_INFORMATION
	GET_ACCESS_CONTROL
	CHANGE_ACCESS_CONTROL
	DELETE_FILE
	RENAME_FILE
	GET_USER_IDENTITY

Module 3 Asynchronous File I/O	Module 4 Multiple Directories
INITIATE_READ	GET_WORKING_DIRECTORY
INITIATE_WRITE	CHANGE_WORKING_DIRECTORY
INITIATE_SEEK	CREATE_DIRECTORY
WAIT_TRANSFER	DELETE_DIRECTORY
ABORT_TRANSFER	GET_DIRECTORY_ENTRY

5.2 Module 1: Device I/O

This module defines a set of functions for generic, low-level access to I/O devices. All devices are viewed as conforming to a common model, and no distinctions are made according to the type of device. Functions that are specific to particular categories of devices may be candidates for definition in future versions of this standard.

This module is not intended to preclude the use of functions from Modules 2 and 3 for access to devices treated as a special case of files, if the implementation supports such access.

The basic model used by this module considers each device to have two inputs and two outputs. The inputs are a *control* input and a *data* input; the outputs are a *status* output and a *data* output. Data I/O is treated as asynchronous; writing control and reading status are regarded as synchronous. The WAIT_DEVICE function, or information obtained from the READ_DEVICE_STATUS function, may be used to determine when a data transfer is complete.

5.2.1 CONNECT_DEVICE

Name: CONNECT_DEVICE (device_name), ({connect_id}, error)

Usage: Creates a binding between the calling process and the device identified by device_name.

Input Items:

device_name. String giving the name of the I/O device to connect to. The format of the name is implementation-dependent.

Returned Items:

connect_id. Systemtag returned by the system that identifies this connection for use with other functions.

Side Effects: None.

Error Conditions:

Class	Reason	Error
3	4	INSUFFICIENT RESOURCES — It was not possible to establish a connection because of physical constraints (e.g., no memory for I/O buffers, no connection identifiers available, a limit on the number of files open, etc.)
3	18	CONNECTION DENIED — The operating system will not grant a connection to this device.
3	36	DEVICE NOT FOUND — There was no device whose name matched the given name.
3	46	DEVICE BUSY — The device was currently unavailable. The operation may be tried again later.

5.2.2 DISCONNECT_DEVICE

Name: DISCONNECT_DEVICE (connect_id), (error)

Usage: Informs the system that the user is finished using this device.

Input Items:

connect_id. Systemtag returned by a previous CONNECT_DEVICE operation.

Returned Items: None.

Side Effects: None.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.

5.2.3 READ_DEVICE_STATUS

Name: READ_DEVICE_STATUS (connect_id, status_type, status_info, status_length), ((amount), error)

Usage: Obtains the status information for the device associated with the specified connection.

Input Items:

connect_id. Systemtag returned by CONNECT_DEVICE identifying the desired device.

status_type. Long integer denoting which status information is desired. The interpretation of this item is implementation-dependent.

status_info. Storageid identifying the place where the desired status information should be placed.

status_length. Integer specifying the maximum amount of status information to be transferred, in bytes.

Returned Items:

amount. Integer denoting the actual amount of information transferred, in bytes.

Side Effects: None.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.
3	14	DATA WAS TRUNCATED — All the data was not transferred.
3	41	INVALID STATUS TYPE — The status type code was not valid for this device or implementation.
3	42	INVALID STATUS LENGTH — The status length value was not valid for this device, status type, or implementation.

5.2.4 WRITE_DEVICE_CONTROL

Name: WRITE_DEVICE_CONTROL (connect_id, control_type, control_info, control_length), ((amount), error)

Usage: Writes control information to the device. This information is normally intended to affect the current state of the device. Examples include selecting printer fonts or emulation modes, setting modem speeds, selecting special modes for a disk drive, etc.

Input Items:

connect_id. Systemtag returned by CONNECT_DEVICE identifying the desired device.

control_type. Long integer denoting which control information is to be written. The interpretation of this item is implementation-dependent.

control_info. Storageid identifying the place where the information to be transmitted to the device resides.

control_length. Integer specifying the amount of control information to be transferred, in bytes.

Returned Items:

amount. Integer denoting the actual amount of information transferred, in bytes.

Side Effects: This function may affect the behavior of the device in a manner that is implementation-dependent.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.
3	14	DATA WAS TRUNCATED — All the data was not transferred.
3	43	INVALID CONTROL TYPE — The control type code was not valid for this device or implementation.
3	44	INVALID CONTROL LENGTH — The control length value was not valid for this device, control type, or implementation.

5.2.5 READ_DEVICE_DATA

Name: READ_DEVICE_DATA (connect_id, data, data_length), ((amount), error)

Usage: Requests transfer of a block of data from a device to the address space of a process. It is the responsibility of the application to determine when and if valid data is available. The status input available from the device may provide a means to determine availability.

Input Items:

connect_id. Systemtag returned by a previous CONNECT_DEVICE operation.

data. Storageid identifying the place to put the transferred data.

data_length. Integer representing the amount of data to transfer, in bytes.

Returned Items:

amount. Integer representing the amount of data actually transferred, in bytes. This value may be less than the number requested if less data remained to be read, or if an error occurred before the transfer was completed.

Side Effects: The data obtained from the device data register or registers is placed into the specified buffer.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.
3	5	ACCESS DENIED — An attempt was made to access the device in a manner not permitted by the operating system.
3	13	TRANSMISSION ERROR — The data could not be transmitted without errors.
3	14	DATA WAS TRUNCATED — All the data was not transferred.
3	35	INVALID DATA LENGTH — The value specified by the <code>data_length</code> input item was not valid for this device or implementation.

5.2.6 WRITE_DEVICE_DATA

Name: WRITE_DEVICE_DATA (`connect_id`, `data`, `data_length`), (`amount`, `error`)

Usage: Requests transfer of a block of data to a device. It is the responsibility of the application to ensure that the device is ready to receive this data. No guarantee is made that the transfer is complete when the function returns. The status input available from the device may provide a means to determine when the device is ready, and when the operation is complete.

Input Items:

`connect_id`. Systemtag returned by a previous CONNECT_DEVICE operation.

`data`. Storageid identifying the data to be transferred.

`data_length`. Integer representing the amount of data to write, in bytes.

Returned Items:

`amount`. Integer representing the amount of data actually transferred, in bytes. This amount may be less than specified if an error occurs.

Side Effects: The data specified is transferred to the device data register or registers.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.

3	5	ACCESS DENIED — An attempt was made to access the device in a manner not permitted by the operating system.
3	13	TRANSMISSION ERROR — The data could not be transmitted without errors.
3	14	DATA WAS TRUNCATED — All the data was not transferred.
3	35	INVALID DATA LENGTH — The value specified by the <code>data_length</code> input item was not valid for this device or implementation.

5.2.7 WAIT_DEVICE

Name: WAIT_DEVICE (`connect_id`, `time_value`, `time_unit`), (*timeout_flag*, *error*)

Usage: Waits up to a specified time for a device to become ready for use. A flag is returned indicating whether the time limit expired before the device was ready.

Input Items:

connect_id. Systemtag returned by a previous CONNECT_DEVICE operation.

time_value. Integer representing the maximum time to wait, as described in 2.5.

time_unit. Integer code representing the units in which the time value is expressed, as described in 2.5.

Returned Items:

timeout_flag. Flag that will have the value *true* if the time limit expired before the device was ready, or *false* otherwise.

Side Effects: None.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.
3	5	ACCESS DENIED — An attempt was made to access the device with an intended access that is not grantable by the operating system.
3	23	INVALID TIME VALUE — The specified time value was not valid.

3 45 INVALID TIME UNIT — The specified time unit was not valid or not supported by this implementation.

5.2.8 GET_DEVICE_CHARACTERISTICS

Name: GET_DEVICE_CHARACTERISTICS (connect_id), ({device_type}, details_block, details_length, error)

Usage: Obtains information about a device.

Input Items:

connect_id. Systemtag returned by CONNECT_DEVICE, specifying the desired device.

Returned Items:

device_type. Integer specifying the device type, encoded as follows:

0	unknown
1	terminal
2	printer
3	hard disk
4	flexible disk
5	D-A converter
6	A-D converter
7	1/2-in magnetic tape
8	1/4-in cartridge tape
9	paper-tape reader
10	paper-tape punch
11	card reader
12	card punch
13	bit-map display
14	vector display
15	modem
16	network device
17	graphic input device
18	memory disk
19-9999	reserved
9999-32 767	user-defined

details_block. Storageid identifying a parameter block that contains more detailed device-specific information.

details_block_length. Integer specifying the number of bytes of information contained in the parameter block.

Side Effects: None.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.

5.3 Module 2: Synchronous File I/O

This module supports a uniform interface to the *files* or logical collections of data maintained by an operating system. In general a file may be a collection of data maintained on a storage device such as a magnetic disk or tape, or it may be a stream of data produced by or sent to an input or output device such as a terminal, printer, communication device, or measurement and control device. Regardless of the physical implementation, all files are viewed in the same way by this module.

The standard recognizes that each particular file system implementation may support file attributes and operations not specified herein, and may not support others that the standard does specify. This diversity of file systems is accommodated as far as possible. Reasonable defaults are specified for features that may not be supported by an implementation. Each file system is of course free to include additional features not covered by this standard.

Files can be accessed either sequentially or in a direct- or random-access manner (if supported by their implementation). A *connection* establishes a logical binding between a process and a file to be used. All the file functions operate on connections. In general, the bulk of system overhead in using a file is made at the time of the connection. Thus, a subsequent OPEN_FILE call has a low system overhead. Multiple connections to the same file are allowed.

The standard preserves the file security of the underlying operating system, but is not overbearing on those systems maintaining minimal security. It defines a file structure with access protection. Systems that do not support such an environment can be mapped to the MOSI interface, implementing only those functions supported and returning appropriate error codes when a specific function is not available.

An *access control* is a permanent attribute of a file authorizing specific operations (e.g., Read, Write, Append) by specific users or groups of users. An *access permission* is an attribute of an open file authorizing specific operations by the current user while the file is open.

This standard recognizes the commonly used access controls and permissions listed in the following table. It does not restrict the use of additional access controls by a particular operating system. For each entry in the table, the first column lists the integer code assigned to the access control. All values are powers of two, and combinations may be formed by adding the values.

1	Delete	Privilege to delete the file.
2	Read	Privilege to read the file.
4	Write	Privilege to write to the file or truncate the file.
8	Extend	Privilege to extend the file.
16	Execute	Privilege to execute the code in the file.
32	Change Access	Privilege to change the access control of the file.

64	Exclusive Access	Privilege to request exclusive access to the file while caller has file opened.
128	Read Lock	Privilege to request that a read lock be set for the file (see 6.2).
256	Write Lock	Privilege to request that a write lock be set for the file (see 6.2).

A *file organization* is a method of organizing the data within a file, which may affect the ways in which the file can be accessed. This standard recognizes the following file organizations:

1	sequential	The file contains data in a linear sequence intended to be accessed only in order.
2	direct	The file contains data that may be accessed randomly by specifying the location of each item.
3	directory	The file contains a directory of other files. Only limited operations are permitted on directories.

A *record organization* is an attribute of the records that comprise a file. The standard recognizes the following record organizations:

1	fixed length	All records contain the same number of bytes.
2	variable length	All records do not contain the same number of bytes.
3	stream	Each record consists of a single byte.

An *access method* is a method of accessing the data in a file. This standard recognizes the following access methods:

1	sequential stream	The data is accessed sequentially. The length of each transfer in bytes is specified by the caller.
2	sequential record	The data is accessed sequentially. The length of each transfer is a specified number of records, the size of which is determined by the operating system and the application program.
3	direct record	The data is accessed randomly. The length of each transfer is a specified number of records, the size of which is determined by the operating system and the application program.

The standard assumes that direct files consist of fixed-length or variable-length records. Fixed-length records may be accessed in multiple-record units, while variable-length records can only be accessed one at a time.

The format of a completely specified, or “fully qualified,” file name is implementation-dependent, but normally includes device name or volume name, directory, and file name with or without extensions. For names not fully qualified, the operating system is assumed to use the current default or working directory (including the applicable device and volume). Functions to examine and set the current default directory name, for those systems supporting multiple directories, are defined in 5.5.

A *temporary file* is one created by the CREATE_FILE function for temporary use. Temporary files may be assigned arbitrary unique names by the operating system. They are deleted automatically when they are disconnected.

A *textfile* is a file that is considered to consist of text characters grouped into lines. A file designated as a textfile may be subject to special treatment by the operating system or application. If the system supports processing textfiles as variable-length record files, then a record is equivalent to a line. In this case, the `READ_FILE` function applied to a textfile with a variable-length record access mode shall perform a read-line operation. This operation will deliver a line with any line-ending codes removed. In similar fashion, a `WRITE_FILE` function applied to a text file with a variable-length record-access mode shall perform a write-line operation. This operation will add any required line-ending codes to each record written.

An attempt to create or change access on a file, with neither *delete* nor *change access* control specified, may result in an undeletable file. This standard does not prohibit this action and the results will be implementation-dependent.

MOSI makes no assumptions about partial buffers or end-of-line or record indicators. The effect of reading an area in a file not previously written is not defined.

5.3.1 CONNECT_FILE

Name: `CONNECT_FILE` (*file_name*), (*connect_id*), *error*)

Usage: Creates a binding between the calling process and the file identified by the specified name.

Input Items:

file_name. String name of the file to connect to. The format of the name is system-dependent.

Returned Items:

connect_id. Systemtag returned by the system that identifies this connection for use with other functions.

Side Effects: None.

Error Conditions:

Class	Reason	Error
3	2	FILE NOT FOUND — There was no file whose name matched the given name.
3	4	INSUFFICIENT RESOURCES — It was not possible to establish a connection because of physical constraints (e.g., no memory for I/O buffers, no connection identifiers available, a limit on the number of files open, etc.).
3	18	CONNECTION DENIED — The operating system would not grant a connection to this file.

5.3.2 DISCONNECT_FILE

Name: DISCONNECT_FILE (connect_id), (error)

Usage: Informs the system that the user is finished using this file. A DISCONNECT_FILE is not necessary if the file was deleted.

Input Items:

connect_id. Systemtag returned by a previous CONNECT_FILE operation.

Returned Items: None.

Side Effects: A DISCONNECT_FILE on an open file causes it to be closed automatically. A DISCONNECT_FILE on a temporary file results in the temporary file being deleted.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.

5.3.3 CREATE_FILE

Name: CREATE_FILE (file_name, file_organization, record_organization, record_size, text_flag, initial_size), ((connect_id), error)

Usage: Creates a file with specified characteristics. The created file is initially connected but not open. Therefore a CONNECT need not be done for a newly created file, but an OPEN is still required for most other functions to use the file. The intended file organization, record organization, and record size are specified for those operating systems that can use this data. No file space is necessarily allocated on creation. Space may be dynamically allocated on writes by some systems, or blocks of space may be allocated by a call to EXTEND_FILE. The file is created with initial access control, which provides complete access to the owner, and no access to anyone else.

Input Items:

file_name. String giving the name of the file to create. The format of the name is implementation-dependent. If a null string is specified, a temporary file will be created. The name for this file will be assigned by the operating system and may be determined using the GET_FILE_INFORMATION function.

file_organization. Integer specifying the file organization (see 5.3). Only code values 1 (sequential) or 2 (direct) are permitted for this item.

record_organization. Integer specifying the record organization (see 5.3).

record_size. Non-negative integer representing the record size in units of bytes for fixed-length record files. For variable-length record files, this item

is interpreted as a maximum limit. A value of 0 means no limit is specified. For sequential stream files, this value must be 1.

text_flag. Flag that has the value *true* if the file is to be interpreted as a textfile, or *false* otherwise.

initial_size. Non-negative long integer specifying the number of records (for a fixed-length record file) or number of bytes (for a stream or variable-length record file) to be initially allocated by systems that require this information. This item may be ignored by systems that allocate file space dynamically.

Returned Items:

connect_id. Systemtag returned by the system that identifies this connection for use with other functions.

Side Effects: The name of the created file is placed in the appropriate directory, with access controls, file organization, record size, and initial allocation indicated.

Note the possibility of a race condition existing between testing via **CONNECT_FILE** for nonexistence and creating via **CREATE_FILE** if it did not exist.

Error Conditions:

Class	Reason	Error
3	3	INVALID FILE NAME — The file name given was not acceptable to the operating system.
3	4	INSUFFICIENT RESOURCES — It was not possible to establish connection because of physical constraints (e.g., no memory for I/O buffers, no connection identifiers available, a limit on the number of files open, etc.).
3	5	ACCESS DENIED — An attempt was made to access the file with an intended access that is not grantable by the operating system for this file.
3	6	INVALID ACCESS CONTROL — Inappropriate access mode for the storage media of the file (read from printer, extend a tape, etc.).
3	10	DUPLICATE FILE NAME — Attempt was made to create a file that already existed.
3	20	FILE ORGANIZATION NOT SUPPORTED — The file organization chosen was not supported by the system.
3	22	UNABLE TO CREATE — The operating system was not able to create the file requested.

3 31 INVALID RECORD ORGANIZATION — The requested record organization was invalid.

5.3.4 DELETE_FILE

Name: DELETE_FILE (connect_id), (error)

Usage: Deletes the specified file and disconnects it. The user must have the delete access control to delete the file.

Input Items:

connect_id. Systemtag returned by a previous CONNECT_FILE or CREATE_FILE operation.

Returned Items: None.

Side Effects: The file is deleted and disconnected. If the file is open it will be closed before deleting. The standard does not specify when the physical deletion of a file occurs.

Error Conditions:

Class	Reason	Error
3	1	INVALID CONNECTION — The connection descriptor was invalid or unknown to the calling process.
3	5	ACCESS DENIED — An attempt was made to access the file with an intended access that is not grantable by the operating system for this file.
3	17	FILE BUSY — The file could not be deleted because of other current users of the file.

5.3.5 EXTEND_FILE

Name: EXTEND_FILE (connect_id, extend_length), (error)

Usage: Allocates additional space to a file. The file must be open. Systems that dynamically allocate space as needed are not required to extend the file when this call is given. No error is signaled by such systems if the file is not extended.

Input Items:

connect_id. Systemtag returned by a previous CREATE_FILE or CONNECT_FILE operation.

extend_length. Non-negative long integer specifying the amount of additional space to allocate to the file. This amount specifies a number of records for fixed-length record files, or a number of bytes for variable-length record files or stream files.