HONEYWELL

LEVEL 68 MULTICS USER RING INPUT/OUTPUT SYSTEM PROGRAM LOGIC MANUAL

RESTRICTED DISTRIBUTION

SOFTWARE

Honeywell

MULTICS USER RING INPUT/OUTPUT SYSTEM PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68)

RESTRICTED DISTRIBUTION

SUBJECT:

Description of the Multics User Ring Input/Output System

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the _Multics Programmers' Manual, Commands and Active Functions_ (Order No. AG92), _Subroutines_ (Order No. AG93), and _Subsystem Writers' Guide_ (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set, which when complete, will supersede the _System Programmers' Supplement to the Multics Programmers' Manual_ (Order No. AK96).

DATE:

May 1977

ORDER NUMBER:

AN57, Rev. 0

PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005

CONTENTS

		Page
Section I	Introduction	1-1
Section II	Design	2-1
	I/O Control Block Visible Portion	2-3
	States of an I/O Control Block	2-5
	I/O Control Block Hidden Portion	2-6
	Common IPS Masking Logic	2-7
Section III	Procedures in the I/O Switching Mechanism	3-1
	find_iocb.pl1	3-1
	iox_	
	$find_iocb	3-1
	$find_iocb_n	3-2
	$look_iocb	3-2
	$destroy_iocb	3-3
	attach_name.pl1	3-4
	iox_	
	$attach_name	3-5
	$attach_ptr	3-6
	iox_.alm	3-6
	iox_	
	$user_io	3-7
	$user_input	3-8
	$user_output	3-8
	$error_output	3-8
	$detach_iocb	3-8
	$close	3-8
	$get_line	3-8
	$get_chars	3-8
	$put_chars	3-8
	$modes	3-8
	$position	3-8
	$control	3-8
	$read_record	3-8
	$write_record	3-8
	$rewrite_record	3-8
	$delete_record	3-8
	$seek_key	3-8
	$read_key	3-8
	$read_length	3-8
	$err_old_dim	3-9
	$err_no_operation	3-9

CONTENTS (cont)

Page

$err_not_attached 3-9
$err_not_closed 3-9
$err_not_open 3-9
propagate.pl1 3-10
iox_
$propagate 3-10
print_attach_table.pl1 3-13
io.pl1 3-13
io 3-14
io_call 3-14

Section IV Design of the IOS Compatibility Package 4-1
Design Characteristics 4-1
Data Structures 4-3

Section V Procedures in the IOS Compatibility Package 5-1
ios_.pl1 5-1
ios_
$attach 5-1
$detach 5-4
$read 5-6
$write 5-7
$abort 5-7
$order 5-8
$resetread 5-8
$setsize 5-9
$getsize 5-9
$setdelim 5-10
$getdelim 5-10
$seek 5-11
$tell 5-12
$changemode 5-12
$readsync 5-13
$writesync 5-14
$no_entry 5-14
$read_ptr 5-15
$write_ptr 5-15
$ios_quick_init 5-16
get_at_entry_.pl1 5-16
get_at_entry_ 5-16
iox_.alm 5-17
iox_
$ios_call 5-18
$ios_call_attach 5-19

CONTENTS (cont)

		Page
	ios_write_around_.pl1	5-19
	netd_attach	5-20
	abs_attach, mr_attach, ocd_attach, tekd_attach, ec_attach	5-21
	ios_write_around_	
	$ios_write_around_detach	5-21
	$ios_write_around_open	5-21
	$ios_write_around_close	5-22
	$ios_write_around_get_line	5-22
	$ios_write_around_get_chars	5-22
	ios_write_around_put_chars	5-22
	ios_write_around_control	5-23
	$ios_write_around_position	5-23
	$ios_write_around_modes	5-23
Section VI	Synonym I/O Module	6-1
	syn_attach.pl1	6-1
	syn_	
	$syn_attach	6-1
	$syn_attach_	6-2
	$syn_detach	6-3
Section VII	Teletype I/O Module	7-1
	tty_.pl1	7-1
	tty_	
	$tty_attach	7-1
	$tty_detach	7-3
	$tty_open	7-3
	$tty_close	7-4
	$tty_get_chars	7-5
	$tty_get_line	7-5
	$tty_put_chars	7-5
	$tty_control	7-6
	$tty_modes	7-6
	$tty_position	7-7
Section VIII	Discard I/O Module	8-1
	discard_attach.pl1	8-1
	discard_	
	$discard_attach	8-1
	$discard_detach	8-1
	$discard_open	8-3
	$discard_close	8-3
Section IX	The vfile_ I/O Module	9-1
	Introduction	9-1
	Program Modules	9-1
	Indexed Files	9-3
	Synchronization of Access	9-6
	Introduction	9-6

CONTENTS (cont)

Page

Data Structures 9-6
Processing 9-7
File Level 9-7
File-Altering Operations 9-8
Passive Index Referencing Operations 9-9
Maintaining Correct Index Position 9-10
Proof of Passive File-Synchronization 9-10
Record Lock Processing 9-12
Record Locking and Unlocking . . 9-12
Passive Record Synchronization 9-13
Recovery From Interruptions 9-14
Introduction 9-14
The Normal State of Update Processing 9-14
Tracking Variables 9-15
Other Header Variables 9-15
The Restart Procedure 9-16
The Repeating State of Execution 9-17
Flow of Control 9-18
Reversion to the Normal State . . 9-19
Restoration of Variables 9-19
Reconstructed Variables 9-20
Protected Variables 9-20
Repeating State-Summary 9-21

Section X Design of the ANSI Standard and IBM Standard Tape I/O Modules 10-1
Introduction 10-1
IOX Functions 10-2
Attach Function 10-2
Open Function 10-3
Close Function 10-3
Detach Function 10-4
Other Functions 10-4
Data Structures 10-5
Control Segment 10-5
Physical I/O Section 10-11
Volume Chain 10-12
File Chain 10-14
File Data Structure (File Data Link) 10-14
File Link 10-17

CONTENTS (cont)

Page

Section XI Procedures in the ANSI Standard and IBM Standard I/O Modules 11-1
tape_ansi_attach_ 11-1
$tape_ansi_attach 11-1
$tape_ibm_attach 11-1
check_attopt 11-6
tape_ansi_file_cntl_ 11-6
$open 11-7
$data_eof 11-12
$data_eot 11-13
$position_for_output 11-14
$beginning_of_file 11-15
$end_of_file 11-16
$close 11-17
$debug_on 11-18
$debug_off 11-18
abort_file 11-19
another_volume 11-20
write_permit 11-20
append_file 11-20
back_TM 11-21
build1 11-21
build2 11-22
build_eofsl 11-22
build_fl 11-22
make_eofsl_real 11-23
consistent 11-23
creating_first 11-23
desired_check 11-24
desired_file 11-24
extend_check 11-25
fill_XXX1 11-26
fill_XXX2 11-26
fill_fl_from_HDR1 11-27
fill_fl_from_HDR2 11-28
fill_fdhdr2_from_fl 11-29
fill_flhdr2_from_fd 11-29
file_new_section_fl 11-29
handler 11-29
initialize_permit 11-30
initialize_permitA 11-30
initialize_volume 11-30
lrec_open 11-31
move_to_EOD 11-31
move_tape_ 11-32
find_candidate 11-34
move_to_first_HDR 11-35
move_forward 11-35
move_backward 11-35
next_volume 11-36

Page

process_EOX 11-36
read_HDR1 11-37
read_HDR2 11-38
read_label 11-38
write_label 11-38
setup_for_create 11-39
setup_for_extend_modify . . . 11-39
setup_for_read 11-40
setup_for_generate 11-40
truncate_chains 11-41
vl_init 11-42
vname 11-42
write_HDRs 11-42
write_EOFs 11-43
write_EOVs 11-43
write_TM 11-43
write_new_section 11-44
tape_ansi_nl_file_cntl_ 11-44
$open 11-44
$data_eof 11-46
$data_eot 11-47
$beginning_of_file 11-48
$end_of_file 11-49
$close 11-50
abort_file 11-51
consistent 11-51
handler 11-52
initialize_permitA 11-52
initialize_permitB 11-52
lrec_open 11-53
move 11-53
next_volume 11-55
vl_init 11-55
vname 11-55
another_volume 11-55
write_TM 11-56
tape_ansi_detach_ 11-56
tape_ansi_lrec_io_ 11-58
$read_record 11-59
get_record 11-63
process_sw 11-64
skip_segments 11-65
move_to_user 11-65
read_release 11-65
tape_ansi_lrec_io_
$write_record 11-66
get_buf 11-71
move_to_buf 11-71
write_buf 11-72

CONTENTS (cont)

```
                                                Page
          tape_ansi_lrec_io_
               $close . . . . . . . . . . . . . . 11-73
          tape_ansi_ibm_lrec_io_
               $read_record . . . . . . . . . . . 11-74
                      get_record  . . . . . . . . 11-78
                      process_sw  . . . . . . . . 11-79
                      skip_segments . . . . . . . 11-80
                      move_to_user  . . . . . . . 11-80
                      read_release  . . . . . . . 11-80
              tape_ansi_ibm_lrec_io_
               $write_record  . . . . . . . . . . 11-81
                      get_buf . . . . . . . . . . 11-87
                      move_to_buf . . . . . . . . 11-87
                      write_buf . . . . . . . . . 11-88
              tape_ansi_ibm_lrec_io_
               $close . . . . . . . . . . . . . . 11-89
          tape_ansi_read_length_  . . . . . . . . 11-90
          tape_ansi_position_ . . . . . . . . . . 11-92
          tape_ansi_mount_cntl_ . . . . . . . . . 11-95
              tape_ansi_mount_cntl_
               $mount . . . . . . . . . . . . . . 11-95
               $remount . . . . . . . . . . . . . 11-96
               $write_rings . . . . . . . . . . . 11-97
               $write_permit  . . . . . . . . . . 11-98
               $free  . . . . . . . . . . . . . . 11-98
                  VOL1_check  . . . . . . . . . . 11-99
          tape_ansi_tape_io_  . . . . . . . . . . 11-100
              tape_ansi_tape_io_
               $attach  . . . . . . . . . . . . . 11-100
               $open  . . . . . . . . . . . . . . 11-101
               $close . . . . . . . . . . . . . . 11-101
               $get_buffer  . . . . . . . . . . . 11-101
               $release_buffer  . . . . . . . . . 11-102
               $read  . . . . . . . . . . . . . . 11-103
               $sync_read . . . . . . . . . . . . 11-103
               $sync_write  . . . . . . . . . . . 11-104
               $write . . . . . . . . . . . . . . 11-105
               $order . . . . . . . . . . . . . . 11-105
          tape_ansi_interpret_status_ . . . . . . 11-107
          tape_ansi_parse_options_  . . . . . . . 11-107
                  no_next . . . . . . . . . . . . 11-109
                  vname . . . . . . . . . . . . . 11-110
          tape_ansi_control_  . . . . . . . . . . 11-110

Section XII   The tape_mult_ I/O Module . . . . . . . . . 12-1
                  Introduction  . . . . . . . . . . . . . 12-1

Section XIII  The tape_nstd_ I/O Module . . . . . . . . . 13-1
                  Introduction  . . . . . . . . . . . . . 13-1
```

SECTION I

INTRODUCTION

The Multics user ring I/O switching mechanism provides a flexible, efficient, and device-independent I/O capability to users and system programs. Explicit Input/Output was not very important in the development of Multics because of its implicit replacement by the virtual memory mechanism. As Multics matured and commercial applications multiplied, as more files grew to exceed segment size, and as demands increased for complete and compatible implementations of the standard I/O interfaces defined for PL/I and COBOL, explicit I/O took on greater significance. Finally, the initial I/O system was replaced by a new one with more flexibility. The old I/O system interfaces are still supported.

The old I/O system is referred to as IOS; its user subroutine interface is ios_. The new I/O system is referred to as IOX; its user subroutine interface is iox_.

For the sake of generality, all input and output with the exception of paging takes place over <u>I/O switches</u>. The requestor of I/O over an I/O switch is generally not aware of the device or data structure to/from which an operation takes place, nor of the procedures that perform the operation. Both devices and procedures are interchangeable in the sense that they support the common interface of the I/O switch. The user acts as though he is doing I/O on the switch itself.

The association of an I/O switch with a device or data structure and with a set of procedures is known as <u>attachment</u>. The device or data structure is known as the <u>target</u> of the attachment. The set of procedures is known as the <u>I/O module</u> serving the attachment. Two operations are defined on an attached I/O switch:

detach	reverse the process of attachment.
open	announce the intention to perform I/O activity of a specified kind.

Operations defined on an open I/O switch are a subset of those listed below, each of which has a standard calling sequence supported by all the I/O modules to which the switch can be attached.

close	reverse the process of opening.
get_line	read the next line of input.
get_chars	read a specified number of characters.
put_chars	write a specified number of characters.
modes	inspect or change the rules governing later I/O operations.
position	space forward or backward without transmitting any data.
control	perform a special I/O operation unique to a particular attachment.
read_record	read the next input record.
write_record	write the next output record.
rewrite_record	replace the record last read or written.
delete_record	discard the last record located.
seek_key	position an indexed file to a specific key.
read_key	read the key to which a file is currently positioned.
read_length	obtain the length of the record to be read next.

An <u>opening mode</u> is a particular way of accessing a file. The opening mode, specified in the open operation, defines a subset of allowed operations on the open I/O switch. Some data bases and some I/O modules do not support all the opening modes, but every module that supports a given opening mode implements all the operations defined for that mode with their system-wide meanings (exceptions are noted in the documentation of individual I/O modules). Table I lists the opening modes in Multics and shows the operations supported by each one.

All opening modes support the close operation and can, depending on the I/O module, support certain modes and control operations. The detach operation is defined only when an I/O switch is closed.

A _synonym_ is an I/O switch attached to another I/O switch directly so that an operation requested on the first switch is performed by the second switch's corresponding operation. Synonyms are very efficient in Multics and should be exploited.

An _I/O control block_ or IOCB is the physical realization of an I/O switch. The IOCB contains information describing the attachment of the I/O switch and an entry variable corresponding to each operation listed above. At attach and open time, the I/O module serving the attachment assigns appropriate entry point values to the entry variables for the operations it supports and assigns entry points that return error codes to the other entry variables. When an I/O operation is requested, the I/O system routes the call to the entry point specified in the IOCB. This routing is done without an extra stack frame or an argument list.

Five I/O modules are a part of the standard system:

discard_	provides a sink for output.
ntape_	performs I/O to/from files on tape.
syn_	establishes one switch as a synonym for another.
tty_	performs I/O to/from terminals.
vfile_	performs I/O to/from files in the storage system.

The user can write his own I/O module and specify it in an attach call. Rules for writing an I/O module are given in Section IV of the _MPM Subsystem Writers' Guide_, Order No. AK92.

This manual describes the operation of the I/O switching mechanism itself and the five standard I/O modules. To find out how to use any of the standard I/O modules, refer to Section III of the _MPM Subroutines_, Order No. AG93. For documentation of the I/O switching mechanism, refer to the iox_ description in Section II of the _MPM Subroutines_.

CONTENTS (cont)

Page

Section XIV The rdisk_ I/O Module 14-1
Introduction 14-1

Section XV The record_stream_ I/O Module 15-1
Introduction 15-1
Program Modules 15-1
record_stream_attach.pl1 15-2
record_stream_attach 15-2
open_rs 15-2
close_rs 15-2
detach_rs 15-2
modes_rs, control_rs 15-3
rs_open_str_in.pl1 15-3
get_chars_rs 15-3
get_line_rs 15-3
position_str_rs 15-3
rs_open_str_out.pl1 15-4
put_chars_rs 15-4
rs_open_seq_in.pl1 15-4
read_record_rs 15-4
read_length_rs 15-4
position_seq_rs 15-5
rs_open_seq_out.pl1 15-5
write_record_rs 15-5

SECTION II

DESIGN

The I/O switching mechanism (as distinguished from the I/O modules that it references) maintains the IOCBs of processes. All data needed for the operation of the I/O system is stored in the IOCBs. There are no other data bases.

A user desiring a new IOCB must call iox_$find_iocb. There are three reasons for centralizing this function within the I/O switching mechanism rather than allowing user programs to allocate their own IOCBs:

1. Only the I/O system itself is guaranteed to be up to date on the current format and required initialization of an IOCB.
2. The I/O system is able to locate all IOCBs at all times.
3. The IOCBs are kept in storage that is least susceptible to accidental damage.

Details of IOCB allocation are discussed under find_iocb.pl1 in Section III of this manual.

An IOCB has two parts, one visible to I/O modules and one hidden from everyone except the I/O switching mechanism. The visible part is the top of the IOCB, referenced by include declarations that show only that part. No user programs or I/O modules need be modified if the hidden portion of the IOCB changes.

The include file used by I/O modules and other user programs to reference an IOCB is iocbv.incl.pl1:

```
dcl 1 iocb aligned based,
     2 iocb_version fixed init(1),
     2 name char(32),
     2 actual_iocb_ptr ptr,
     2 attach_descrip_ptr ptr,
     2 attach_data_ptr ptr,
     2 open_descrip_ptr ptr,
     2 open_data_ptr ptr,
     2 reserved bit(72),
2 detach_iocb entry(ptr,fixed(35)),
2 open entry(ptr,fixed,bit(1)aligned,fixed(35)),
2 close entry(ptr,fixed(35)),
2 get_line entry(ptr,ptr,fixed(21),fixed(21),fixed(35)),
2 get_chars entry(ptr,ptr,fixed(21),fixed(21),fixed(35)),
2 put_chars entry(ptr,ptr,fixed(21),fixed(35)),
2 modes entry(ptr,char(*),char(*),fixed(35)),
2 position entry(ptr,fixed,fixed(21),fixed(35)),
2 control entry(ptr,char(*),ptr,fixed(35)),
2 read_record entry(ptr,ptr,fixed(21),fixed(21),fixed(35)),
2 write_record entry(ptr,ptr,fixed(21),fixed(35)),
2 rewrite_record entry(ptr,ptr,fixed(21),fixed(35)),
2 delete_record entry(ptr,fixed(35)),
2 seek_key entry(ptr,char(256)varying,fixed(21),fixed(35)),
2 read_key entry(ptr,char(256)varying,fixed(21),fixed(35)),
2 read_length entry(ptr,fixed(21),fixed(35));
```

A second include file, for ALM assemblies, shows both visible and hidden portions. This file is iocbs.incl.alm:

```
bool     iocb.version,0       .. 1 ..     fixed
bool     iocb.name,1                      char (32)
bool     iocb.actual_iocb_ptr, 12         ptr
bool     iocb.attach_descrip_ptr,14       ptr
bool     iocb.attach_data_ptr,16          ptr
bool     iocb.open_descrip_ptr,20         ptr
bool     iocb.open_data_ptr,22            ptr
bool     iocb.event_channel,24            bit (72)
bool     iocb.detach_iocb,26              entry
bool     iocb.open,32                     entry
bool     iocb.close,36                    entry
bool     iocb.get_line,42                 entry
bool     iocb.get_chars,46                entry
bool     iocb.put_chars,52                entry
bool     iocb.modes,56                    entry
bool     iocb.position,62                 entry
bool     iocb.control,66                  entry
bool     iocb.read_record,72              entry
bool     iocb.write_record,76             entry
bool     iocb.rewrite_record,102          entry
bool     iocb.delete_record,106           entry
bool     iocb.seek_key,112                entry
bool     iocb.read_key,116                entry
bool     iocb.read_length,122             entry

Hidden information, to support SYN attachments.

bool     iocb.ios_compatibility,126       ptr
bool     iocb.syn_inhibits,130            bit (36)
bool     iocb.syn_father,132              ptr
bool     iocb.syn_brother,134             ptr
bool     iocb.syn_son,136                 ptr
```

I/O CONTROL BLOCK VISIBLE PORTION

The first field is the version number. The current version number is 1. Because of the expense of reprogramming all the existing I/O modules, the only changes expected to be made to the visible portion of the IOCB are additions to the end, moving the hidden information down.

The second field is the 32-character name of the IOCB. This name is assigned at the time the IOCB is created and never changes.

The third field is the actual_iocb_ptr, which points to the IOCBs actual IOCB. An IOCB that is not a synonym of another is its own actual IOCB. The actual_iocb_ptr of a synonymed IOCB is the same as the actual_iocb_ptr of the IOCB to which it is synonymed. A set of IOCBs synonymed to each other form a tree, of which the actual IOCB is the root. The actual_iocb_ptr is manipulated by the synonym I/O module, syn_attach, when synonyms are attached and detached. Whenever syn_attach modifies an IOCB, it also modifies the IOCBs actual IOCB. It then calls iox_$propagate, which propagates the change to all members of the synonym tree.

The I/O system is organized so that changes to IOCBs are comparatively infrequent, occurring mainly at attach, detach, open, and close times. The cost of having synonyms is paid at these times rather than when a IOCB is used. This cost consists of a call to iox_$propagate after making any change.

The fourth field is the attach_descrip_ptr. By definition, if this pointer is null, the IOCB is detached. The IOCB is attached if attach_descrip_ptr is not null, in which case this pointer points to a varying character string called the <u>attach description string</u>. This is a human- and machine-readable description of the IOCBs attachment. It is fabricated by the I/O module serving the attachment from the information passed to it in the attach call. The attach description string is a sequence of arguments separated by blanks. The first argument is the name of the I/O module serving the attachment. The second is the name of the target of the attachment. Any remaining arguments are attachment-specific options whose format is known only to the particular I/O modules that support them.

An attach description string contains all the information needed to complete an attachment. The I/O system provides two entry points that accept attach description strings and perform the described attachments. See iox_$attach_ptr and iox_$attach_name in Section III of this manual.

Attach description strings form a convenient interface for PL/I programmers to specify the default attachments of PL/I files. If the PL/I runtime I/O support finds that the IOCB associated with a PL/I file is not attached when I/O is requested on it, the PL/I title of the file is used as an attach description string to perform the attachment.

The attach description string of an IOCB synonymed to another IOCB is a description of the synonymization rather than a copy of the other IOCBs attach description string. This field and two others, name and attach_data_ptr, retain their individual identity in a synonym attachment. All other fields in the visible portion of a synonymed IOCB are duplicate copies of the corresponding fields of the actual IOCB.

The print_attach_table (pat) command prints the attach description strings of specified switches or of all the attached switches in the process.

The fifth field is the attach_data_ptr. The I/O module serving the attachment can use this pointer to locate a data structure of its own design in which it remembers whatever details of the attachment it needs in whatever format is appropriate. The contents of this data structure are the exclusive property of the serving I/O module. An I/O module is not allowed to keep data pertaining to a particular attachment anywhere but in or locatable from the attach data structure pointed to by attach_data_ptr. This requirement is enforced so that I/O operations on one attachment have no effect upon the state of another attachment through the same I/O module.

The sixth field is the open_descrip_ptr. By definition, if this pointer is null, the IOCB is closed. The IOCB is open if open_descrip_ptr is not null, in which case this pointer points to a varying character string called the <u>open description string</u>. This is a human- and machine-readable description of the opening of the IOCB, fabricated by the serving I/O module from the information passed to it in the open call. The open description string is a sequence of arguments separated by blanks. The first argument is the name of the opening mode. Any remaining arguments are attachment-specific.

The seventh field is the open_data_ptr. Like attach_data_ptr, this pointer is used by the serving I/O module to locate a data structure describing the current opening.

The eighth field, named reserved in the include file iocbv.incl.pl1, is presently not used. It provides space for an event channel id to support asynchronous I/O.

The remaining sixteen fields in the visible portion of the IOCB, detach_iocb through read_length, are entry variables into which the serving I/O module stores the values of entry points in itself that implement the I/O operations. In detached and closed IOCBs, these entry variables are set by the I/O switching mechanism to entry points in the I/O switching mechanism that do nothing but return error codes. Three such error codes are error_table_$not_attached, error_table_$not_open and error_table_$no_operation. In open blocks, any entry variable not set by the serving I/O module is set by iox_$propagate to one of the code-returning entry points.

STATES OF AN I/O CONTROL BLOCK

An IOCB is always in one of four states: detached, attached and closed, open, or attached as a synonym. For each of these states, there are certain consistency relationships that must hold among the various fields of the IOCB. The I/O switching mechanism enforces these relationships, first by creating IOCBs in a consistent state, and second by calling iox_$propagate after making each change.

By definition, an IOCB is detached if its attach_descrip_ptr is null. In this state, the I/O system ensures that actual_iocb_ptr points to the IOCB itself, attach_data_ptr, open_descrip_ptr, and open_data_ptr are null, and all entry variables are set to entry points that return error codes.

By definition, an IOCB is attached as a synonym if actual_iocb_ptr points to a different IOCB. In this state, attach_descrip_ptr points to a description of the synonymization as prepared by the synonym module. The field attach_data_ptr is used by the synonym module for its own purposes, and all other fields of the IOCB are duplicate copies of those in the actual IOCB. The actual IOCB must be in either the detached, closed, or open state.

By definition, an actual IOCB is closed if it is attached but open_descrip_ptr is null. In this state, open_data_ptr must be null, the detach_iocb entry variable must be set to an entry point within the I/O module capable of restoring the IOCB to the detached state, the open entry variable must be set to an entry point within the I/O module capable of changing the IOCB to the open state, and all other entry variables must be set to entry points that return error codes.

Finally, by definition, an actual IOCB is open if its open_descrip_ptr is nonnull. In this state, the detach_iocb and open entry variables must be set to entry points that return error codes. All other entry variables are set as desired by the I/O module. Entry values for operations not supported by the particular opening mode remain set to entry points that return error codes.

I/O CONTROL BLOCK HIDDEN PORTION

The hidden portion of an IOCB is used by the I/O switching mechanism to implement synonyms and IOS compatibility.

The first field, ios_compatibility, is a pointer to a module that simulates all of the functions of the obsolete I/O switching mechanism IOS. IOS compatibility is documented in Section IV of this manual.

The remaining fields are used to implement synonyms. These pointers are null if the IOCB is not involved in a synonym attachment. Otherwise, they contain all the information needed by iox_$propagate to find the members of the synonym tree and to propagate the entry variables in the actual IOCB to all the other members.

The bit string syn_inhibits specifies which operations are inhibited by the synonym attachment. There is a one-to-one correspondence between the first fifteen bits of syn_inhibits and the operations open through read_length. There is no bit corresponding to the detach_iocb operation, which cannot be inhibited. When a bit is on, the corresponding operation is inhibited and returns the code error_table_$no_operation. The operation is automatically inhibited for any IOCB synonymed to the given IOCB. When copying the entry variables from an IOCB to one of its descendants, iox_$propagate substitutes an entry point that returns the above error code for the entry values of inhibited operations.

The pointer syn_father points to the IOCB to which the current IOCB is immediately synonymed. This pointer is different from actual_iocb_ptr, which points to the IOCB to which the current IOCB is ultimately synonymed.

The pointer syn_brother points to the next IOCB immediately synonymed to the same father as the current IOCB. This pointer is null if there is no such IOCB.

The pointer syn_son points to the first IOCB immediately synonymed to the current block. All others also immediately synonymed can be located by following the syn_brother thread from the syn son.

COMMON IPS MASKING LOGIC

There are many places in the I/O system where, for short periods of time, loss of control due to an IPS interrupt (quit, alrm, or cput) cannot be tolerated because some IOCBs are in an inconsistent state. All IPS interrupts are masked off while these critical sections are running. The strategy used is:

```
        :
    ips_mask = 0;
    call default_handler_$set(handler);
        :
    call hcs_$set_ips_mask(0,ips_mask);
    CRITICAL SECTION
    call hcs_$reset_ips_mask(ips_mask,ips_mask);
        :

    handler: proc (p1, name, p2, p3, continue);
              if ips_mask^=0 then TERMINATE PROCESS;
              if name^="cleanup" then continue = "1"b;
    end handler;
```

Because faults can still occur while interrupts are masked, a handler must be established to catch them. Any fault that occurs during a critical section must terminate the process because I/O will be disabled in unpredictable ways. However, faults that occur elsewhere are passed on up the stack to the program that normally handles them. The logic shown above uses the automatic variable ips_mask to determine whether control is inside a critical section. This variable must be initialized to zero before enabling the condition handler. Thereafter, a call to hcs_$set_ips_mask upon entry to a critical section saves the current interrupt mask in ips_mask. The last bit is guaranteed to be on, therefore ips_mask is nonzero. A call to hcs_$reset_ips_mask upon leaving the critical section resets ips_mask to zero.

SECTION III

PROCEDURES IN THE I/O SWITCHING MECHANISM

The I/O switching mechanism consists of seven modules:

find_iocb.pl1
attach_ioname.pl1
move_attach.pl1
iox_.alm
propagate.pl1
print_attach_table.pl1
io.pl1

MODULE find_iocb.pl1

This module implements the entry points iox_$find_iocb, iox_$find_iocb_n, iox_$look_iocb, and iox_$destroy_iocb.

This is the only module that manages the allocation and deallocation of IOCBs. All other modules in the I/O system work with IOCBs to which they have been given pointers, these pointers having ultimately been obtained from find_iocb.

Entry: iox_$find_iocb

This entry point returns a pointer to the named IOCB. The IOCB is created if it does not already exist.

Usage

```
dcl iox_$find_iocb entry(char(*),ptr,fixed bin(35));

call iox_$find_iocb (ioname, iocb_ptr, code);
```

1. ioname   is the name of an IOCB. (Input)

2. iocb_ptr is a pointer to the IOCB. (Output)

3. code is a standard status code. (Output)

Entry: iox_$find_iocb_n

This entry point returns a pointer to the nth IOCB allocated in the process.

Usage

```
        dcl iox_$find_iocb_n entry(fixed bin(17),
                                   ptr,fixed bin(35));

        call iox_$find_iocb_n (iocb_n, iocb_ptr, code);
```

1. iocb_n is the number of an IOCB. (Input)

2. iocb_ptr is a pointer to the IOCB. (Output)

3. code is a standard status code. (Output)

The entry point iox_$find_iocb_n is used to methodically locate all the IOCBs in existence, including those that are detached. The IOCBs are numbered contiguously from one through the highest-numbered IOCB. The numbers bear no relation to the order in which the IOCBs were created or used. If a number below one or above the highest number is requested, iox_$find_iocb_n returns a null pointer and the code error_table_$no_iocb.

Creation and destruction of IOCBs changes the numbering. Therefore, a program should not call iox_$find_iocb or iox_$destroy_iocb while using iox_$find_iocb_n to look at all IOCBs.

Entry: iox_$look_iocb

This entry point is the same as iox_$find_iocb but does not create an IOCB.

Usage

```
        dcl iox_$look_iocb entry(char(*),ptr,fixed bin(35));

        call iox_$look_iocb (ioname, iocb_ptr, code);
```

Arguments are the same as for iox_$find_iocb.

Entry: iox_$destroy_iocb

This entry point destroys an IOCB.

Usage

```
        dcl iox_$destroy_iocb entry(ptr,code);

        call iox_$destroy_iocb (iocb_ptr, code);
```

1. iocb_ptr   is a pointer to an IOCB. (Input)

2. code   is a standard status code. (Output)

The caller is responsible for ensuring that the IOCB is in the detached state prior to requesting its destruction, and for guaranteeing that no pointers to it remain afterwards. The storage occupied by the IOCB is made available for other use. If a new IOCB is created with the same name as the old one, it may or may not occupy the same storage.

Internal Logic

Five IOCBs are allocated at one time as a compromise between speed and storage consumption. The first array of five IOCBs occupies the internal static of find_iocb. When the first array has been used up, arrays of five are allocated in the combined linkage segment of the ring in which they reside. The arrays are threaded together and the beginning of the thread is kept in internal static.

The first array of five IOCBs allows room for the four standard system I/O switches, user_i/o, user_input, user_output, and error_output, and one user-defined I/O switch. The IOCBs for the system switches are assigned and initialized to the detached state upon the first call to iox_$find_iocb. Their locations are filled into the external pointer variables iox_$user_io, iox_$user_input, iox_$user_output, and iox_$error_output. These entry points allow fast access to the system switches by all programs.

When the user requests that an IOCB be destroyed, the IOCB is marked free so that it can be reused. An IOCB is marked free by setting its actual_iocb_ptr to null. The allocated storage itself is never freed, but free IOCBs are used before allocating another array.

Maintenance of the IOCB table is slightly complicated by the facts that:

1. The table cannot be locked. If a later invocation of find_iocb needs access to the table while an earlier one has it locked, a deadlock exists.

2. IPS interrupts cannot be masked off for the whole time it takes to search the potentially large table.

Hence, the following strategy is used. Any program that changes the table records the time of the change in the internal static variable changed_at. Any program that searches the table records the current time in the automatic variable searched_at. If at the end of the search it finds that a modification was made during the search (changed_at is greater than searched_at), it repeats the search. The search itself is carefully coded so that it cannot blow up if the table is modified while it is in progress. The modification code is careful not to thread a new array of IOCBs until they have been initialized properly.

The usual I/O system IPS interrupt masking logic is used.

Whenever a new IOCB is assigned, it is initialized to the detached state. The assignment of an IOCB, its initialization, and the setting of the caller's iocb_ptr are all done in one critical section with IPS interrupts masked. Therefore, the assignment of an IOCB is an atomic operation from the standpoint of process synchronization.

Whenever an IOCB is destroyed, it is initialized to the detached state. Therefore, if any I/O operation is attempted on the destroyed IOCB, an error occurs rather than unpredictable I/O. If, however, the same storage is subsequently assigned, pointers to the old IOCB now point to the new IOCB. There is no way to catch this error. The caller must be responsible for setting all copies of iocb_ptr to null when an IOCB is destroyed. The unassignment of the IOCB, its initialization, and the resetting of the caller's iocb_ptr are all done in one critical section with IPS interrupts masked.

MODULE attach_name.pl1

This module implements the entry points iox_$attach_name and iox_$attach_ptr. It attaches an IOCB according to an attach description string.

Entry: iox_$attach_name

This entry point attaches an IOCB whose name is known.

Usage

```
        dcl iox_$attach_name entry(char(*),ptr,
                                     char(*),ptr,fixed   bin(35));

        call  iox_$attach_name  (name,  iocb_ptr, string, ref_ptr,
code);
```

1. ioname — is the name of an IOCB. (Input)

2. iocb_ptr — is a pointer to the IOCB. (Output)

3. string — is an attach description string specifying the desired attachment. (Input)

4. ref_ptr — is null or a pointer to the referencing procedure, used to implement the "referencing_dir" search rule in searching for the I/O module. (Input)

5. code — is a standard status code. (Output)

The code returned is zero if attachment is successful. It is error_table_$no_iocb if no IOCB could be created. It is any error code returned by expand_path_, hcs_$initiate_count, or hcs_$make_ptr if the specified I/O module cannot be initiated, or any error code returned by the I/O module itself if it cannot perform the attachment.

The entry point in the I/O module that is called to perform the attachment is determined as follows. If the module name in the attach description string is not a pathname (does not contain < or >), attach_ioname calls hcs_$make_ptr with the module name as reference name and an entry point name consisting of the module name concatenated with the string "attach" (For example, syn_$syn_attach). If the supplied module name is a pathname, the I/O module is initiated with the module name as a reference name and then hcs_$make_ptr is called as above.

Entry: iox_$attach_ptr

This entry point attaches an IOCB given a pointer to it.

Usage

```
     dcl iox_$attach_ptr entry(ptr,char(*),ptr,fixed  bin(35));

     call iox_$attach_ptr (iocb_ptr, string, ref_ptr, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. string is an attach description string. (Input)

3. ref_ptr is null or a pointer to the referencing procedure. (Input)

4. code is a standard status code. (Output)

This entry point behaves the same as iox_$attach_name.

Internal Logic

The only significant processing done by this module is the parsing of the attach description string into an array of varying character string arguments as required by the I/O module. The parsing is done in two stages. The first parse scans the string to determine how many arguments it contains and the length of the longest one. Space is grown in automatic storage to hold an array of varying strings of that length. The second parse rescans the string, filling in the array elements.

If the attach description string contains no target and no options, attach_name constructs an array of extent zero. Although an array of extent zero is nonstandard PL/I, our implementation of PL/I allows this to happen and handles it correctly.

The I/O module to perform the attachment is called via cu_$ptr_call because its name is not known at compile time.

MODULE iox_.alm

This module implements the following entry points:

iox_$user_io	iox_$write_record
iox_$user_input	iox_$rewrite_record
iox_$user_output	iox_$delete_record
iox_$error_output	iox_$seek_key
iox_$detach_iocb	iox_$read_key
iox_$open	iox_$read_length
iox_$close	iox_$err_no_iocb

iox_$get_line	iox_$err_no_operation
iox_$get_chars	iox_$err_not_attached
iox_$put_chars	iox_$err_not_closed
iox_$modes	iox_$err_not_open
iox_$position	iox_$err_old_dim
iox_$control	iox_$ios_call
iox_$read_record	iox_$ios_call_attach

The iox_ module performs an assortment of functions each of which is best coded in machine language. The first four entry points listed above, iox_$user_io through iox_$error_output, are not executable procedures but external pointer variables that point to the IOCBs for the four system I/O switches. Since most I/O in the system is conducted over these switches, these entry points eliminate many calls to iox_$find_iocb. The next sixteen entry points, iox_$detach_iocb through iox_$read_length, are call forwarders that merely pass control to the corresponding entry variables in the IOCB. They permit most callers of the I/O system to remain unaware of the format of an IOCB. The next six entry points, iox_$err_no_iocb through iox_$err_old_dim, are procedures that do nothing but return a specified error code as the last argument with which they are called. They are used to fill in entry variables that should not be called. The last two entry points, iox_$ios_call and iox_$ios_call_attach, are used only by the IOS compatibility package to perform operations with an obsolete format of I/O module, the interface of such calls having required the use of machine language. Whereas entry points in the module ios_ (described in Section IV) test the format of an IOCB, these last two entry points assume it is of the old format.

Entry: iox_$user_io

This entry point is an external pointer variable that always points to the IOCB for the switch user_i/o.

Usage

```
     dcl iox_$user_io pointer external;

     call iox_$put_chars (iox_$user_io, addr(buf),
                                      length(buf), code);
```

or:

```
     call iox_$user_io->iocb.put_chars(iox_$user_io,
                                      addr(buf),length(buf),code);
```

The external pointer is set during process initialization when user_real_init_admin_ calls iox_$find_iocb for the first time to set up the system I/O switches.

Entries: iox_$user_input

iox_$user_output
iox_$error_output

These three external pointer variables point to the IOCBs for the switches user_input, user_output, and error_output respectively. They are used in the same way as iox_$user_io.

Entry: iox_$detach_iocb

Usage

```
dcl iox_$detach_iocb entry(ptr,fixed bin(35));

call iox_$detach_iocb (iocb_ptr, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. code is a standard status code. (Output)

This entry point passes the call on, with the same argument list, to the entry variable iocb.detach_iocb in the IOCB. The effect is equivalent to executing the call:

```
call iocb_ptr->iocb.detach_iocb (iocb_ptr, code);
```

RESTRICTION: The entry variable in the IOCB must be external. The call-forwarding code in iox_$detach_iocb does not properly pass the environment pointer portion of an internal entry variable.

Entries:

iox_$close	iox_$get_line
iox_$get_chars	iox_$put_chars
iox_$modes	iox_$position
iox_$control	iox_$read_record
iox_$write_record	iox_$rewrite_record
iox_$delete_record	iox_$seek_key
iox_$read_key	iox_$read_length

These entry points are the same as iox_$detach_iocb, differing only in the entry variable to which they pass their calls. The same RESTRICTION applies.

Internal Logic

The call-forwarding code is:

```
detach_iocb:
     epplp     ap|2,*                   Get addr of IOCB ptr.
     epplp     lp|0,*                   Get addr of IOCB itself.
     callsp    lp|iocb.detach_iocb,*    Pass call on.
```

Only the lp register is changed. The callsp instruction is used instead of a transfer because it is possible for the entry value to be a gate. The argument list is not copied or altered in any way. To support calls upon entry variables that take internal procedure values would require copying the argument list to make space for insertion of the environment pointer, which would in turn necessitate a stack push. These steps add too much overhead to every I/O call to justify their inclusion. Hence the above RESTRICTION.

Entry: iox_$err_no_iocb

This entry point returns the status code error_table_$no_iocb as its last argument.

Usage

```
dcl iox_$err_no_iocb entry options(variable);

iocb_ptr->iocb.get_line = iox_$err_no_iocb;
```

The last argument is set to error_table_$no_iocb only if it is declared as fixed bin(35) If iox_$err_no_iocb is called with no arguments, it returns without doing anything. Descriptors are not inspected. This entry point is intended to be placed in an entry variable during periods in which the corresponding operation is disabled.

Entries:

```
iox_$err_no_operation     iox_$err_not_attached
iox_$err_not_closed       iox_$err_not_open
iox_$err_old_dim
```

These entry points behave the same as iox_$err_no_iocb except that they return the status codes error_table_$no_operation, error_table_$not_attached, error_table_$not_closed, error_table_$not_open, and error_table_$old_dim respectively.

Entries: iox_$ios_call and

iox_$ios_call_attach

These entry points form part of the IOS compatibility package, described in Section IV of this manual. They are grouped in iox_.alm only to avoid needless fragmentation at the source module level.

MODULE propagate.pl1

This module implements the entry point iox_$propagate. It is intended to be called by any I/O module after making any change to an IOCB. It has two functions:

1. To make sure the IOCB is in a consistent state.

2. To propagate changes to all IOCBs that are synonyms of the changed IOCB.

In general, I/O modules ignore synonyms, trusting that iox_$propagate will keep them up to date.

While an I/O module is changing the contents of an IOCB, the IOCB can be in an inconsistent state. It can remain in an inconsistent state until iox_$propagate has finished with it. Therefore, the I/O module is responsible for masking off all IPS interrupts before changing the IOCB and keeping them masked off until iox_$propagate returns. It is recommended that the I/O module use the common IPS interrupt masking logic used elsewhere in the I/O switching mechanism. The propagate module itself does not change the IPS interrupt mask or have a handler for IPS interrupts.

Entry: iox_$propagate

Usage

```
dcl iox_$propagate entry(ptr);

call iox_$propagate (iocb_ptr);
```

where iocb_ptr is a pointer to an IOCB. (Input)

No status code is returned by this entry point, first because iox_$propagate has already done its best to make the IOCB consistent and second because there is no meaningful use that the caller could make of such a code.

Internal Logic

Synonym attachments cause IOCBs to be threaded together in tree structures by threads maintained in the IOCBs. There are four pointer fields used for this purpose: iocb.actual_iocb_ptr, iocb.syn_father, iocb.syn_son, and iocb.syn_brother. Of these, only actual_iocb_ptr is in the visible or advertised portion of the IOCB. The meanings of the four threads are as follows:

If an IOCB is a synonym, its syn_father pointer points to the IOCB to which it is immediately synonymed. Otherwise, syn_father is null.

If an IOCB is synonymed to another, its syn_brother pointer points to the next IOCB also synonymed to the other IOCB. If there is no next IOCB so synonymed, or if the given IOCB is not synonymed, syn_brother is null.

If an IOCB is the target of a synonym, syn_son points to the first IOCB synonymed to it. Otherwise, syn_son is null. The remaining IOCBs synonymed to the given IOCB can be found by following the syn_brother thread beginning in its syn_son IOCB.

If an IOCB is synonymed to another, actual_iocb_ptr points to the nonsynonymed IOCB to which it is ultimately synonymed. Otherwise, actual_iocb_ptr points to the given block itself.

The first task performed by iox_$propagate is to ensure the consistency of the given IOCB. If the IOCB is synonymed to another, then the synonym I/O module is trusted to have left the IOCB consistent. If the IOCB is not synonymed to another, it is policed as described in the next four paragraphs.

If the IOCB is detached (attach_descrip_ptr is null), then attach_data_ptr, open_descrip_ptr, open_data_ptr, and ios_compatibility are set to null, event_channel is set to zero, the detach_iocb, open, modes and control entry variables are set to iox_$err_not_attached, and all other entry variables are set to iox_$err_not_open. The entry variables of a detached IOCB are never equal to iox_$err_old_dim because the IOS compatibility package is careful to reset such entries to iox_$err_not_open before calling iox_$propagate.

If the block is attached but not open (attach_descrip_ptr is not null but open_descrip_ptr is null), then open_data_ptr is set to null and all entry variables from get_line on, with the exception of modes and control, are set to iox_$err_not_open. If modes and control are equal to iox_$err_no_operation or iox_$err_not_attached, they are set to iox_$err_not_open. No entry will be equal to iox_$err_old_dim, because the IOS compatibility package never leaves IOCBs attached but not open.

If the IOCB is both attached and open (attach_descrip_ptr and open_descrip_ptr are not null), then the entry variables detach_iocb and open, if not equal to iox_$err_old_dim, are set to iox_$err_not_closed and the entry variables get_line through read_length, if equal to iox_$err_not_open, are set to iox_$err_no_operation. Thus, in the case of IOCBs managed by the IOS compatibility package, the detach_iocb and open entries remain set to iox_$err_old_dim and all other entries are set to supported operations or iox_$err_old_dim. In all other cases, the detach_iocb and open entries are forced to iox_$err_not_closed and all other entries that the I/O module has left equal to iox_$err_not_open are reset to iox_$err_no_operation.

This policing strategy ensures that IOCBs are always in a well-defined state. It also permits I/O modules to concern themselves only with I/O operations of which they wish to be aware. The propagate module automatically keeps untouched fields of the IOCB up to date as its state changes.

Having been forced to be consistent, the fields of the given IOCB must be propagated to the IOCBs that are synonyms of it. A loop does the following for each IOCB synonymed to the given IOCB, i.e., each syn son, found by chasing the syn_son and syn_brother threads:

1. Copies selected fields of the given IOCB to the son.

2. Calls propagate$recurse, an internal entry point, on the son to propagate the changes to all of its syn sons. This call is made only if the son has a syn son.

The entry point propagate$recurse is only called if one of the syn sons of the IOCB on which iox_$propagate was called has a syn son.

Field propagation involves the following. Fields actual_iocb_ptr, open_descrip_ptr, open_data_ptr, event_channel, and ios_compatibility are copied from the given IOCB to the son. For each entry variable from open through read_length, if the corresponding operation is inhibited in the syn attachment of the son to the given IOCB (the corresponding bit is on in the son's syn_inhibits field), the entry variable is set to iox_$err_no_operation. Otherwise, the entry variable is copied from the given IOCB to the son. No change is made to the fields name, attach_descrip_ptr, attach_data_ptr, and syn_inhibits, all of which retain their original identity in synonymed IOCBs.

MODULE print_attach_table.pl1

This module implements the print_attach_table (pat) command, which prints information summarizing the attachment of selected IOCBs or all IOCBs.

If one or more I/O switch names are given, the command calls iox_$look_iocb with each name to get a pointer to the IOCB and calls the internal procedure show to print a line of information. If no names are given, the command calls iox_$find_iocb_n to enumerate all IOCBs and calls show for each.

The internal procedure show formats and prints a one-line (occasionally overflowing to two) display showing the attachment and open status of an IOCB.

MODULE io.pl1

This module implements the io or io_call command, which provides a convenient command-level interface to the operations of the I/O system.

Names: io, io_call

Usage

```
      io  opname  ioname  -control_args-
```

1. opname   is the operation to be performed.

2. ioname   is the name of an I/O switch.

Internal Logic

The IOCB is located via iox_$look_iocb unless ioname is "attach" or "find_iocb", in which case iox_$find_iocb is called. Control is dispatched to the block of code that completes the particular operation.

Two internal subroutines, announce_ and error_, are used by most of the operations to handle errors. Their calling sequences are the same as that of com_err_. Either subroutine returns silently if the status code passed to it is zero. If the code is nonzero, either subroutine calls com_err_, passing on its argument list. If the status code is -1, it is changed to zero and com_err_ is called. The announce_ subroutine returns to its caller afterwards, whereas error_ aborts execution of the io command. The announce_ subroutine is used to report the outcome of the I/O operation and error_ is used to report errors in the execution of the io command.

The IOCB is declared using an old include file named iocbx.incl.pl1, which declares the entry variables with the returns option. Calls to the entry variables are embedded in calls to announce_. The detach operation, for example, says:

```
  call announce_(p->iocb.detach_iocb(p),io_detach,
                                    "^a",p->iocb.name);
```

where p is a pointer to the IOCB. The first argument to announce_ is the code returned by iocb.detach_iocb and the other arguments format the error message, if any.

The attach coding parses an attach description string into an array of varying character string arguments. First, the string is scanned to determine how many arguments it contains and the length of the longest one. Space is grown in automatic storage to hold an array of varying strings of that length. The array is filled in.

The attach coding requests the I/O module to print error messages as well as return a status code. This is done to provide information in addition to the switch name that cannot be conveyed in a status code.

The get_line coding handles the case where the number of characters is not specified, by calling iocb.get_line to read 64 characters each time until the code error_table_$long_record is not returned.

When the number of characters is specified for get_line, get_chars and read_record, the stack is grown to hold a buffer of that size.

The control coding grows stack space for a character string if one is given and passes a pointer to the string as an information pointer. If no string is given to the command, a null information pointer is passed. When iocb.control returns, the string is inspected to see if it has been changed by the I/O module. If it has and the length of the string is less than 200, the string is printed. Otherwise, the value of the information pointer is printed.

The print_iocb coding calls an internal subroutine, piocb, to print the contents of an IOCB. Most fields are printed only if nonzero or nonnull to minimize the amount of output. The entry variables are treated as a based array. When looping through the entry variables, an inner loop skips over successive entries that are identical so that the value of all these entries is printed only once. The complete pathnames of the entry variables are obtained from hcs_$fs_get_path_name and get_entry_name_.

SECTION IV

DESIGN OF THE IOS COMPATIBILITY PACKAGE

The present I/O switching mechanism IOX replaces a previous mechanism known as IOS. Since IOS I/O modules still exist, IOX provides three-way compatibility with IOS:

1. Users can call IOX to perform I/O over switches served by IOS I/O modules.

2. Users can call IOS to perform I/O over switches served by IOX I/O modules.

3. Users can still call IOS to perform I/O over switches served by IOS I/O modules.

Complete compatibility is available for the nine old standard system I/O modules syn, tw_, ntw_, absentee_dim_, mrd_, oc_, tek_, exec_com_, and discard_output_. Either IOS or IOX can be called to attach and detach these modules. For other I/O modules not specifically designed to be compatible, attach and detach must be performed through IOS entry points but all other operations can be performed via IOX.

When all references to the old interfaces are deleted from the system, this compatibility package will be obsolete and can also be deleted.

DESIGN CHARACTERISTICS

Compatibility with IOS necessitates that all calls other than the normal ones from IOX requestor to IOX module are detected and intercepted. Such calls must be mapped into calls of similar function that can be performed by the serving I/O module.

Calls to IOS are easy to detect because they name entry points in the module ios_. The technique for capturing these calls is to provide a new implementation of ios_. The new implementation ascertains whether the intended I/O operation is directed to an I/O switch served by an IOX module or an IOS module and maps the call accordingly. It must be aware of how old and new module attachments are represented in the new I/O switching mechanism.

Calls to IOX naming I/O switches served by IOS modules can be caught as follows: Since IOS modules do not use the entry variables in the IOCB, IOX can fill them in with intercept routines when it attaches the IOS module. The intercept routines map IOX calls into functionally equivalent operations that can be performed by the serving IOS module.

Attachments served by new and old I/O modules are distinguished by the ios_compatibility pointer in the hidden portion of the control block. This pointer is null for an attachment served by an IOX I/O module. For an attachment served by an IOS I/O module, ios_compatibility points to the module's transfer vector. (Each old I/O module contains a transfer vector that serves the same purpose as the array of entry variables in the current I/O switching mechanism.)

Entry points in ios_ note from a null ios_compatibility pointer that the attachment is served by an IOX module and call entry variables in the IOCB. The entry variable called is chosen to carry out the intended IOS operation. The mapping of an IOS request into its corresponding IOX operation is discussed in detail under ios_.pl1 in Section V of this manual.

The attachment of an IOCB to a serving IOS I/O module is represented as follows: The location of the module's transfer vector is stored in ios_compatibility. The location of the module's stream data block, the old system's equivalent to the open_data_block, is stored in open_data_ptr. These are the only items necessary to support IOS calls on the module because the stream data block contains the equivalent of the attach and open descriptions and the transfer vector replaces the array of entry variables. The remainder of the IOCB is filled in to support IOX calls: The attach description string contains the name of the I/O module and the name of the target. The open description string contains the opening mode stream_input_output followed by the IOS mode designation specified in the IOS attach call. The entry variables of the IOCB are initialized to write-around routines that create the appearance of the IOX mode stream_input_output by means of IOS calls on the serving IOS module. These write-arounds are discussed in detail under ios_write_around_ in Section V of this manual.

DATA STRUCTURES

The IOCB is the only data structure used by the IOS compatibility package.

SECTION V

PROCEDURES IN THE IOS COMPATIBILITY PACKAGE

IOS compatibility is implemented by four modules: ios_.pl1, get_at_entry_.pl1, iox_.alm, and ios_write_around_.pl1.

MODULE ios_.pl1

This module implements the following entry points:

ios_$attach	ios_$getdelim
ios_$detach	ios_$seek
ios_$read	ios_$tell
ios_$write	ios_$changemode
ios_$abort	ios_$readsync
ios_$order	ios_$writesync
ios_$resetread	ios_$no_entry
ios_$resetwrite	ios_$read_ptr
ios_$setsize	ios_$write_ptr
ios_$getsize	ios_$ios_quick_init
ios_$setdelim	

All entry points except ios_$attach and ios_$detach are write-arounds that forward their calls to the appropriate entry points in the I/O module serving the attachment.

<u>Entry</u>: ios_$attach

<u>Usage</u>

```
     dcl ios_$attach entry(char(*),char(*),char(*),
                       char(*),bit(72)aligned);

     call ios_$attach (switch, dim, device, mode, status);
```

1. switch	is the name of an I/O switch. (Input)
2. dim	is the reference name of an I/O module. (Input)
3. device	is the name of the intended target of the attachment. This argument can specify a logical device, a volume identifier, another switch name, the name of a file, etc., depending on the serving I/O module. (Input)
4. mode	is a string defining the initial mode to be assumed by the I/O module if it has changeable modes. If mode is "", the default mode for the I/O module is assumed. (Input)
5. status	is an IOS-style status code: standard status code in the first 36 bits, status flags in the last 36 bits. (Output)

This entry point is called by programs using the obsolete I/O switching mechanism IOS in order to perform the attach operation. The attachment can be served by either an IOS module or an IOX module. In either case, the attachment is such that both IOS and IOX calls can be made on the I/O switch.

If the I/O switch named by the switch argument is already attached and not as a synonym, ios_$attach immediately returns the error code error_table_$ionmat.

If the switch is attached as a synonym, it is provisionally detached. The attachment status is saved for restoration in the event that an error prevents the new attachment. If the new attachment fails, the old attachment is restored.

The dim argument is inspected to see if it is the name of one of the nine old standard system I/O modules syn, tw_, ntw_, absentee_dim_, mrd_, oc_, tek_, exec_com_, and discard_output_. For these modules, the attach call is forwarded to one of the new I/O modules syn_, tty_, netd_, abs_, mr_, ocd_, tekd_, ec_, and discard_.

If this is a synonym attachment, processing ends here. For other attachments, the switch is opened (iox_$open) with the mode stream_input_output (stream_output in the case of discard_output_) and a modes operation is performed (iox_$modes) with the mode argument given to ios_$attach.

INTERNAL LOGIC

The IOS compatibility package maintains the integrity of IOCBs at all times by proper use of the common I/O system IPS interrupt masking strategy. To attach via an IOX I/O module, ios_ masks all IPS interrupts, verifies that the IOCB in question is detached (or detaches it if it is synonymed), attaches via a call to the I/O module, restores the old synonym attachment if the new attachment has failed, and restores IPS interrupts.

Ideally, the same sequence should be followed when attaching an IOS I/O module, but old I/O modules are not designed to run with interrupts masked. The following circuitous route is taken instead. IPS interrupts are masked while the attachment state of the IOCB is being inspected. If the IOCB is detached, interrupts are restored and the attach entry point of the I/O module is called. The I/O module does not change the IOCB to record the attachment. When the I/O module returns, the IOCB must be checked again in case it has been attached in the meantime by another process. If it has not, ios_ records the new attachment in the IOCB. If it has been attached by another process, ios_ returns an error code. In the latter case, it is possible that the I/O module has obtained the use of reserved resources such as tape drives or special communications lines. These resources are not made available again until the process terminates.

To attach an IOS I/O module, designed to be called by a special machine language calling sequence, ios_$attach uses the entry points iox_$ios_call and iox_$ios_call_attach. These entry points, described below, are PL/I-callable write-arounds written in machine language.

When the attachment of an IOX I/O module is requested through ios_$attach, the resulting attachment is indistinguishable from an attachment performed by IOX. The contents of the IOCB are exactly the same. IOX calls use the entry variables stored there by the I/O module. IOS calls are mapped by the individual ios_ entry points into corresponding IOX calls. The logic for each mapping is described below under the appropriate ios_ entry point.

An IOCB served by an IOS module is organized as follows: The attach description string constructed by ios_$attach is of the form "<dim> <device>", where <dim> and <device> are the values of the input arguments dim and device. The open description string is of the form "IOS compatibility <mode>", where <mode> is the value of the mode argument. The attach_data_ptr points to a structure in which the attach and open description strings are stored. The open_data_pointer is the stream data block pointer received from the IOS I/O module's attach call. The ios_compatibility pointer points to the transfer vector of the IOS I/O module. The entry variables get_line through control are filled in with write-arounds that convert IOX calls to IOS calls. The entry variable close is set to a procedure that performs detachment. Therefore, detachment is performed automatically by an IOX close operation. All other entry variables are set to iox_$err_old_dim, meaning that these operations are not supported by an IOS I/O module.

IOS calls on such an IOCB are handled by the various ios_ entry points described below. In general, these entry points check whether the ios_compatibility pointer is null and if it is not, use iox_$ios_call to call the correct entry point in the I/O module's transfer vector.

The attach data block created by ios_$attach is used by the write-arounds for get_line through control, which assume that this structure begins with copies of the old I/O module's stream data block pointer and transfer vector pointer. The format of the attach data block constructed by ios_$attach must not be changed without considering how such a change would affect the various write-arounds.

<u>Entry</u>: ios_$detach

<u>Usage</u>

```
     dcl ios_$detach entry(char(*),char(*),char(*),
                                           bit(72)aligned);

     call ios_$detach (switch, device, disposal, status);
```

1. switch	is the name of an IOCB. (Input)
2. device	is the name of the target to which the IOCB is supposedly attached. (Input)
3. disposal	is optional information passed on to the I/O module that affects details of the detachment, such as retention of demountable volumes. (Input)
4. status	is an IOS-style status code. (Output)

This entry point can be used to undo any attachment, whether attached via an IOS call or an IOX call and whether served by an IOS module or an IOX module.

INTERNAL LOGIC

IPS interrupts are masked and the IOCB is inspected to determine the proper course of detachment.

First, if the IOCB is not found or is not attached, ios_$detach immediately returns the error code error_table_$ioname_not_found.

Second, if the IOCB is attached as a synonym, ios_$detach calls the detach entry variable in the IOCB and turns on the detached-status bit in the status argument. The synonym I/O module adjusts the IOCB to the detached state. The check for a synonym attachment must be made before the check for an old or new I/O module so that an IOCB synonymed by IOX to an IOCB attached by IOS is not mistaken to be attached by IOS.

Third, if the IOCB is attached through an IOX I/O module, it is closed if open by calling its close entry variable and then detached by calling its detach entry variable. If no errors have occurred, ios_$detach turns on the detached-status bit in the status argument.

Fourth and last, if the IOCB is attached through an IOS I/O module, IPS interrupts are restored and detachment is performed. The IOCB must be inspected afterwards to make sure it has not been attached or detached in the meantime by another process. If it has, ios_$detach immediately returns the error code error_table_$ionmat without changing the IOCB. Otherwise, it changes the IOCB to the detached state.

REMAINING IOS ENTRIES

The entry points ios_$read through ios_$writesync utilize an internal procedure named setup. This procedure does preliminary work common to all of these operations.

First, setup clears the caller's status argument. It calls iox_$look_iocb to get a pointer to the IOCB associated with the caller's switch argument, setting the variable iocb_ptr. If the IOCB is not found, not attached, or not open, setup places an appropriate error code in status and aborts the I/O operation by a nonlocal goto to a return statement.

The entry points ios_$read through ios_$writesync use the following common logic:

```
operation:  entry (switch,...,status);
            call setup();
            if iocb_ptr->iocb.ios_compatibility=null then do;
                NEW I/O MODULE CODE
            end;
            else do;
                OLD I/O MODULE CODE
            end;
            return;
```

The setup call sets iocb_ptr. The IOCBs ios_compatibility pointer determines whether the attachment is served by an IOX module or an IOS module, and the appropriate action is taken in each case.

Entry: ios_$read

Usage

```
dcl ios_$read entry(char(*),ptr,fixed bin,fixed bin,
                    fixed bin,bit(72)aligned);

call ios_$read (switch, bufptr, offset, nelem,
                nelemt, status);
```

1. switch	is the name of an IOCB. (Input)
2. bufptr	is a pointer to a buffer to be used in the transmission. (Input)
3. offset	is the offset within the buffer at which to place the first element read. (Input)
4. nelem	is the number of elements requested to be read. (Input)
5. nelemt	is the number of elements actually read. (Output)
6. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the read slot of the module's transfer vector.

If the attachment is served by an IOX I/O module, the call is transformed into a get_line operation specifying the same offset and number of elements. This transformation is meaningful only if the element size is one character. Any status code returned by get_line except error_table_$long_record is passed back to the caller of ios_$read.

Entry: ios_$write

Usage

```
        dcl ios_$write entry(char(*),ptr,fixed bin,fixed bin,
                                 fixed bin,bit(72)aligned);

        call ios_$write (switch, bufptr, offset, nelem,
                                 nelemt, status);
```

where arguments are the same as for ios_$read, above.

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the write slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is transformed into a put_chars operation specifying the same offset and number of elements. This transformation is meaningful only if the element size is one character. If the module returns a zero status code, nelemt is set equal to nelem because IOX modules always transmit all the characters given them. Otherwise, nelemt is set to zero. All status codes are passed back to the caller of ios_$write.

Entry: ios_$abort

Usage

```
        dcl ios_$abort entry(char(*),bit(72)aligned,
                                 bit(72)aligned);

        call ios_$abort (switch, unused, status);
```

1. switch	is the name of an IOCB. (Input)
2. unused	is unused.
3. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the abort slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is transformed into a control operation specifying the order "abort". Any status code returned by control is passed back to the caller. IOX modules desiring to implement the abort operation do so by recognizing the "abort" order in a control operation.

Entry: ios_$order

Usage

```
dcl ios_$order entry(char(*),char(*),ptr,bit(72)aligned);

call ios_$order (switch, order, info_ptr, status);
```

1. switch	is the name of an IOCB. (Input)
2. order	is the name of the control function to be performed. (Input)
3. info_ptr	is a pointer to an optional data structure required by certain orders. The data itself can be input or output or both. (Input)
4. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the order slot of the module's transfer vector.

If the attachment is served by an IOX I/O module, the call is forwarded without change to the module's control operation.

Entry: ios_$resetread

Usage

```
dcl ios_$resetread entry(char(*),bit(72)aligned);

call ios_$resetread (switch, status);
```

where arguments 1 and 2 are the same as for ios_$order.

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the resetread slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is transformed into a control operation specifying the order "resetread". Any status code returned by resetread is passed back to the caller. IOX modules desiring to implement the resetread operation do so by recognizing the "resetread" order in a control operation.

Entry: ios_$setsize

Usage

```
        dcl  ios_$setsize entry(char(*),fixed bin,bit(72)aligned);

        call ios_$setsize (switch, elemsize, status);
```

1. switch	is the name of an IOCB. (Input)
2. elemsize	is the desired element size in bits. (Input)
3. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the setsize slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the setsize operation. None of the system modules specifically converted to IOX supports this operation.

Entry: ios_$getsize

Usage

```
        dcl ios_$getsize entry(char(*),fixed bin,bit(72)aligned);

        call ios_$getsize (switch, elemsize, status);
```

1. switch	is the name of an IOCB. (Input)
2. elemsize	is the current element size of the attachment in bits. (Output)
3. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the getsize slot of the module's transfer vector.

If the attachment is served by an IOX module, an elemsize of nine is automatically returned. This element size is the only one supported by IOX modules.

Entry: ios_$setdelim

Usage

```
     dcl ios_$setdelim entry(char(*),fixed bin,(*)bit(*),
                       fixed bin,(*)bit(*),bit(72)aligned);

     call ios_$setdelim (switch, nbreaks, breaklist,
                         ndelims, delimlist, status);
```

1. switch   is the name of an IOCB. (Input)

2. nbreaks   is the number of break elements supplied. (Input)

3. breaklist is an array of break elements, each of the current element size. (Input)

4. ndelims   is the number of delimiter elements supplied. (Input)

5. delimlist is an array of delimiter elements, each of the current element size. (Input)

6. status   is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the setdelim slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table_$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the setdelim operation.

Entry: ios_$getdelim

Usage

```
     dcl ios_$getdelim entry(char(*),fixed bin,(*)bit(*),
                        fixed bin,(*)bit(*),bit(72)aligned);

     call ios_$getdelim (switch, nbreaks, breaklist,
                         ndelims, delimlist, status);
```

1. switch   is the name of an IOCB. (Input)

2. nbreaks   is the number of break elements currently in use for this attachment. (Output)

3. breaklist	is an array of the break elements currently in use for this attachment, each of the current element size. (Output)
4. ndelims	is the number of delimiter elements currently in use for this attachment. (Output)
5. delimlist	is an array of the delimiter elements currently in use for this attachment, each of the current element size. (Output)
6. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the getdelim slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table_$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the getdelim operation.

Entry: ios_$seek

Usage

```
     dcl ios_$seek entry(char(*),char(*),char(*),
                         fixed bin,bit(72)aligned);

     call ios_$seek (switch, name1, name2,
                         offset, status);
```

1. switch	is the name of an IOCB. (Input)
2. name1	is the name of an old I/O system reference pointer whose value is to be changed. (Input)
3. name2	is the name of an old I/O system reference pointer whose value, incremented by offset number of elements, is to be assigned to the pointer named by name1. (Input)
4. offset	is the number of elements used to compute the value assigned to the reference pointer named by name1. (Input)
5. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the seek slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table_$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the seek operation.

Entry: ios_$tell

Usage

```
dcl ios_$tell entry(char(*),char(*),char(*),
                        fixed bin,bit(72)aligned);

call ios_$tell (switch, name1, name2,
                        offset, status);
```

1. switch	is the name of an IOCB. (Input)
2. name1	is the name of the old I/O system reference pointer whose value is desired. (Input)
3. name2	is the name of an old I/O system reference pointer. (Input)
4. offset	is the number of elements by which the value of the reference pointer named by name1 exceeds the value of the reference pointer named by name2. (Output)
5. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the tell slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table_$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the tell operation.

Entry: ios_$changemode

Usage

```
dcl ios_$changemode entry(char(*),char(*),char(*),
                                bit(72)aligned);

call ios_$changemode (switch, newmode, oldmode, status);
```

1. switch	is the name of an IOCB. (Input)

2. newmode	is a character string describing modes to be established for subsequent I/O operations. (Input)
3. oldmode	is a character string describing the modes that were previously in effect. (Output)
4. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the changemode slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is forwarded unchanged as a modes operation.

<u>Entry</u>: ios_$readsync

<u>Usage</u>

```
dcl ios_$readsync entry(char(*),char(*),char(*),
                                bit(72)aligned);

call ios_$readsync (switch, mode, offset, status);
```

1. switch	is the name of an IOCB. (Input)
2. mode	is a character string indicating whether the read synchronization mode is to become synchronous or asynchronous. (Input)
3. offset	is the maximum number of elements that the I/O module is permitted to read ahead in the asynchronous mode. (Input)
4. status	is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the readsync slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table_$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the readsync operation. None of the nine system modules converted to IOX allows any control over synchronization.

Entry: ios_$writesync

Usage
```
        dcl ios_$writesync entry(char(*),char(*),fixed bin,
                                 bit(72)aligned);

        call ios_$writesync (switch, mode, offset, status);
```

1. switch — is the name of an IOCB. (Input)

2. mode — is a character string indicating whether the write synchronization mode is to become synchronous or asynchronous. (Input)

3. offset — is the maximum number of elements that the I/O module is permitted to write behind in the asynchronous mode. (Input)

4. status — is an IOS-style status code. (Output)

If the attachment of switch is served by an IOS I/O module, the call is forwarded to the writesync slot of the module's transfer vector.

If the attachment is served by an IOX module, the call is rejected with the error code error_table_$missent. That is, ios_ assumes that an attachment served by an IOX module does not support the writesync operation.

Entry: ios_$no_entry

Usage
```
        dcl ios_$no_entry entry options(variable);
```

This entry point, when called with an argument list whose last argument is an IOS-style status code, sets that code to error_table_$missent and returns. It is intended to be placed in slots of transfer vectors corresponding to I/O operations that are not supported by a given module. It ensures that, should such an operation be requested, the caller receives an appropriate error code.

This entry point is not intended to be called directly.

Entry: ios_$read_ptr

This entry point is a handy abbreviation for the get_line operation on the I/O switch user_input.

Usage

```
     dcl ios_$read_ptr entry(ptr,fixed bin,fixed bin);

     call ios_$read_ptr (bufptr, nelem, nelemt);
```

1. bufptr	is a pointer to a buffer to be used in the transmission. (Input)
2. nelem	is the number of elements requested to be read. (Input)
3. nelemt	is the number of elements actually read. (Output)

The call is transformed into a get_line operation on user_input. The error codes error_table_$long_record and error_table_$end_of_info are ignored. All other status codes from get_line are passed to ios_signal_ for reporting. If ios_signal_ returns, the get_line operation is repeated.

Entry: ios_$write_ptr

This entry point is a handy abbreviation for the put_chars operation on the I/O switch user_output.

Usage

```
     dcl ios_$write_ptr entry(ptr,fixed bin,fixed bin);

     call ios_$write_ptr (bufptr, offset, nelem);
```

1. bufptr	is a pointer to the data to be written. (Input)
2. offset	is the number of the first element to be written. (Input)
3. nelem	is the number of elements to be written. (Input)

The call is transformed into a put_chars operation on user_output. All status codes returned by put_chars are passed to ios_signal_ for reporting. If ios_signal_ returns, the put_chars operation is repeated.

Entry: ios_$ios_quick_init

This entry point initializes the standard system I/O switches.

Usage

```
        dcl ios_$ios_quick_init entry;

        call ios_$ios_quick_init;
```

The call establishes user_input, user_output, and error_output as synonyms for user_i/o. Any errors terminate the process with the message "Unable to perform necessary attachments."

This entry point is called by user_real_init_admin_ early in the life of each process.

MODULE get_at_entry_.pl1

This module implements the entry point get_at_entry_, an external interface in the IOS I/O system for acquiring certain information about an I/O attachment.

Entry: get_at_entry_

Usage

```
       dcl get_at_entry_ entry(char(*),char(*),char(*),
                                     char(*),fixed bin(35));

       call get_at_entry_ (switch, dim, device, mode, code);
```

1. switch	is the name of an IOCB. (Input)
2. dim	is the name of the I/O module serving the attachment. (Output)
3. device	is the name of the target of the attachment. (Output)
4. mode	is a character string indicating the modes currently in effect for the attachment. (Output)
5. code	is a standard status code. (Output)

Internal Logic

Since all attachments in the new I/O system are maintained in IOCBs, get_at_entry_ has been rewritten to extract the same information as before but from different sources. Three cases are distinguished internally.

The first case is that of an attachment served by an IOS I/O module. This case is distinguished by a nonnull ios_compatibility pointer in an IOCB that is an actual IOCB. In this case, get_at_entry_ returns the same information as it did in the old I/O system.

The second case applies to the nine system modules that have been explicitly converted to IOX. A list of the nine new and nine old names is maintained by get_at_entry_. In this case, the same information is returned as in the old I/O system.

The third case is that of an attachment served by an IOX I/O module of unknown name. This case has no counterpart in the old I/O system, so the information returned by get_at_entry_ is an approximation. The I/O module name and target name are taken from the attach description string. A modes operation is attempted to obtain the modes. If it fails, blanks are returned for mode.

MODULE iox_.alm

In addition to implementing entry points of the new I/O system, this module also implements the entry points iox_$ios_call and iox_$ios_call_attach as part of the IOS compatibility package. These two entry points are used by PL/I callers to call old I/O modules, which have special calling sequences involving the use of index register six and a transfer vector.

Entry: iox_$ios_call

Usage

```
        dcl iox_$ios_call entry options(variable);
        dcl 1 ics aligned,
             2 sdbptr ptr,
             2 dimptr ptr,
             2 offset fixed bin;
        dcl status bit(72) aligned;

        ics.sdbptr = ...;
        ics.dimptr = ...;
        ics.offset = ...;
        call iox_$ios_call (addr(ics), ... ,status);
```

This entry point is used to call all operation entries of an IOS I/O module except the attach operation. The caller must first fill in the IOS-caller communication structure, ics. The first item in ics is the stream data block pointer for the attachment. The compatibility package stores this pointer in the open_data_ptr field of the IOCB. The next item in ics is a pointer to the module's transfer vector. The compatibility package stores this pointer in the ios_compatibility field of the IOCB. The third item in ics is the offset in the transfer vector of the entry to be called. The offset assignments for the various I/O operations are:

1	detach	9	getsize
2	read	10	setdelim
3	write	11	getdelim
4	abort	12	seek
5	order	13	tell
6	resetread	14	changemode
7	resetwrite	19	readsync
8	setsize	20	writesync

This entry point is called with the same calling sequence as the corresponding ios_ entry point except that a pointer to the ics structure is passed in place of the switch name.

Internal Logic

The argument list is inspected to find the last argument (status) and set it to zero. Many I/O modules assume that status has been cleared before they are called. The first argument pointer, which points to a pointer to the ics structure, is changed to point to ics itself. The desired offset is loaded into index register six and control is transferred to the module's transfer vector, located via the second item in ics. No stack frame is created, and when the module's I/O operation completes, it returns directly to the caller of iox_$ios_call.

Entry: iox_$ios_call_attach

Usage

```
        dcl iox_$ios_call_attach entry(char(*),char(*),
                     char(*),char(*),bit(72)aligned,ptr);

        call iox_$ios_call_attach (switch, dim,
                     device, mode, status, addr(ics));
```

where arguments 1-5 are the same as for iox_$ios_attach.

This entry point is used to call the attach entry point of an IOS I/O module. Its calling sequence is similar to that of iox_$ios_call. The differences are: 1) ics.offset need not be filled in, and 2) a pointer to the ics structure is passed as an additional argument rather than in place of the switch name.

MODULE ios_write_around_.pl1

This module implements IOX I/O operations on six of the converted I/O modules by building IOS-compatible control blocks and forming calls to iox_$ios_call. This module (or the bound segment containing it) has the names netd_, abs_, mr_, ocd_, tekd_ and ec_. The entry points that handle the various I/O operations are:

ios_write_around_detach	detach_iocb
ios_write_around_open	open
ios_write_around_close	close
ios_write_around_get_line	get_line
ios_write_around_get_chars	get_chars
ios_write_around_put_chars	put_chars
ios_write_around_control	control
ios_write_around_position	position
ios_write_around_modes	modes

There is also an attach entry point for each I/O module. The entry name called by iox_$attach_iocb for each of the converted I/O modules is formed by concatenating the new I/O module name with the string "attach", for example, netd_$netd_attach. The attach and open operations fill the IOCB with the values of the above entry points in ios_write_around_.

Entry: netd_attach

This entry point attaches an IOCB via write-arounds by setting the values of the open and detach_iocb entry variables to the entry points ios_write_around_open and ios_write_around_detach.

Usage

```
     dcl netd_$netd_attach entry(ptr,(*)char(*),
                             bit(1)aligned,fixed bin(35));

     call netd_$netd_attach (iocb_ptr, args,
                                 loud_sw, code);
```

1. iocb_ptr	is a pointer to an IOCB. (Input)
2. args	is an array of extent one where args(1) is the attach description string. (Input)
3. loud_sw	if ON, means print an error message on the user's terminal before returning a nonzero status code. (Input)
4. code	is a standard status code. (Output)

Internal Logic

Common I/O system IPS masking strategy is used. The internal procedure error_ is used to report any errors. This internal procedure restores IPS interrupts if they are masked, prints a message on the terminal if loud_sw is on, and returns via a nonlocal goto to a return statement.

If the IOCB is already attached, the attach operation is aborted by calling error_. The device name is taken from args(1), the attach description string, and passed to iox_$ios_call_attach. The ics structure contains a null sdbptr and ics.dimptr points to the transfer vector for the actual IOS module, in this case ntw_$ntw_module. The I/O module name passed to iox_$ios_call is the old I/O module name, in this case ntw_.

If the attachment is not successful, netd_attach calls error_. Otherwise, it prepares to record the attachment in the block by allocating a stream data block or sdb. The attach_descrip_ptr in the IOCB is set to point to the attach description in the sdb and attach_data_ptr is set to point to the sdb itself. The entry variables detach_iocb and open are set to the entry points ios_write_around_detach and ios_write_around_open in ios_write_around_ so that the appropriate write-arounds will perform these operations. Finally, iox_$propagate is called and IPS interrupts are restored.

Entries: abs_attach, mr_attach, ocd_attach,

tekd_attach, ec_attach

These entry points have the same calling sequences and internal operation as netd_attach. The old I/O module name and transfer vector name for each is listed below:

I/O module	old name	transfer vector
abs_	absentee_dim_	absentee_dim_$absentee_dim_module
mr_	mrd_	mrd_$mrd_module
ocd_	oc_	oc_$oc_module
tekd_	tek_	tek_$tek_module
ec_	exec_com_	exec_com_$exec_com_module

Entry: ios_write_around_detach

The calling sequence is the same as for iox_$detach_iocb. Common IPS interrupt masking strategy is used. If the detach_iocb entry variable in the IOCB is not equal to this entry point, as for a call from outside the I/O switching mechanism, the entry point in the IOCB is called instead.

An ics structure is built from the information in the stream data block pointed to by attach_data_ptr. The detach call is made via iox_$ios_call. If detachment is successful, the IOCB is set to the detached state.

Entry: ios_write_around_open

The calling sequence is the same as for iox_$open. Common IPS interrupt masking strategy is used. If the open entry variable in the IOCB is not equal to this entry point, the entry point in the IOCB is called instead.

The open description is determined from the mode argument and the string "-extend" is appended to it if the extend bit argument is on. Appropriate entry points in ios_write_around_ are filled into the IOCB.

No call is made to iox_$ios_call because there is no IOS operation corresponding to the IOX open operation.

Entry: ios_write_around_close

The calling sequence is the same as for iox_$close. Common IPS interrupt masking strategy is used. If the close entry variable in the IOCB is not equal to this entry point, the entry point in the IOCB is called instead.

The detach_iocb and open entry variables in the IOCB are set to ios_write_around_detach and ios_write_around_open and other entry variables are set to iox_$err_not_open.

Each of the following entry points builds an ics structure from information in the stream data block pointed to by attach_descrip_ptr.

Entry: ios_write_around_get_line

The calling sequence is the same as for iox_$get_line.

The caller-supplied buffer pointer is separated into a word-aligned pointer and an offset and iox_$ios_call is called. If zero elements are returned, ios_write_around_get_line returns error_table_$end_of_info. If the last of the returned elements is not a newline character, ios_write_around_get_line returns error_table_$long_record.

Entry: ios_write_around_get_chars

The calling sequence is the same as for iox_$get_chars.

Repeated calls to iox_$ios_call read 64 elements at a time until the desired number of elements is read or error_table_$end_of_info is returned.

Entry: ios_write_around_put_chars

The calling sequence is the same as for iox_$put_chars.

The caller-supplied buffer pointer is split into a word-aligned pointer and an offset and iox_$ios_call is called to perform the operation.

Entry: ios_write_around_control

The calling sequence is the same as for iox_$control.

The value of ics.entry and the nature of the iox_$ios_call call depends on the order argument. An IOX control operation is used to perform a variety of IOS operations, depending on the value of order.

Entry: position

The calling sequence is the same as for iox_$position.

The device is positioned 128 elements at a time by repeated calls to iox_$ios_call.

Entry: ios_write_around_modes

The calling sequence is the same as for iox_$position.

The value of ics.entry is set equal to 14, the IOS offset for changemode, and iox_$ios_call is called.

SECTION VI

SYNONYM I/O MODULE

MODULE syn_attach.pl1

This module implements synonym attachment and detachment. There are two attach entry points, syn_attach and syn_attach_. The first is the normal attach entry. The second is called by ios_ to restore a synonym attachment when the attachment intended to replace it has failed.

<u>Entry</u>: syn_$syn_attach

<u>Usage</u>

```
     dcl syn_$syn_attach entry(ptr,(*)char(*),
                                bit(1),fixed bin(35));

     call syn_$syn_attach (iocb_ptr, args, loud_sw, code);
```

1. iocb_ptr	is a pointer to an IOCB. (Input)
2. args	is an array of arguments. The first argument is the name of the target of the synonymization. The second argument can be "-inhibit" or "-inh", in which case the succeeding arguments are the names of inhibited operations. (Input)
3. loud_sw	is ON if an error message is to be printed on the user's terminal before returning a nonzero status code. (Input)
4. code	is a standard status code. (Output)

Internal Logic

Common IPS interrupt masking strategy is used. Errors are handled by the internal procedure error_, which prints a message if loud_sw is on and aborts the attach operation by a nonlocal goto to a return statement.

A pointer to the target switch is obtained by calling iox_$find_iocb on args(1). The remaining arguments are inspected to build an inhibits bit string identical to syn_inhibits in the IOCB. From this point on, attachment proceeds the same as for syn_$syn_attach_ below.

Entry: syn_$syn_attach_

Usage

```
dcl syn_$syn_attach_ entry(ptr,ptr,bit(*),fixed  bin(35));

call syn_$syn_attach_ (iocb_ptr, target_ptr,
                                  inhibits, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. target_ptr is a pointer to the target of the old synonym attachment to be restored. This pointer is the old iocb.syn_father. (Input)

3. inhibits is the syn_inhibits bit string from the old attachment. (Input)

4. code is a standard status code. (Output)

Internal Logic

Common IPS interrupt masking strategy and the error_ internal procedure are used.

A data block called blk is allocated in the linkage section for each attachment. These blocks are threaded together and have the following format:

```
dcl 1 blk aligned based(blkptr),
      2 next ptr,
      2 attach char(189) varying;
```

An attach description string is built from the target name and the string "-inh" if any operations are inhibited followed by the names of the inhibited operations. The attach_descrip_ptr in the IOCB is set to point to blk.attach and attach_data_ptr to blk itself. Entry variables for inhibited operations are set to iox_$err_no_operation and all others are copied from the target IOCB. The syn_father pointer is set to point to the target and syn_brother to the target's syn son. The target's syn_son pointer is set to point to the IOCB being attached. Changes are propagated by iox_$propagate.

Entry: syn_$syn_detach

Usage

```
dcl syn_$syn_detach entry(ptr,fixed bin(35));

call syn_$syn_detach (iocb_ptr, code);
```

Internal Logic

Common IPS interrupt masking strategy is used. If the detach_iocb entry variable in the IOCB is not equal to this entry point, the entry point in the IOCB is called instead. The data block blk is rethreaded from the used list to a free list where it can be assigned to another attachment. The IOCB is set to the detached state. If there are syn brothers, the detached IOCB is removed and the list of syn brothers is rethreaded. Changes are propagated by iox_$propagate.

SECTION VII

TELETYPE I/O MODULE

MODULE tty_.pl1

This module implements all I/O operations on printing terminals by calling appropriate hardcore system entries.

Entry: tty_$tty_attach

This entry point performs terminal attachment, entailing the creation of an event channel.

Usage

```
     dcl tty_$tty_attach entry(ptr,(*)char(*),
                               bit(1),fixed bin(35));

     call tty_$tty_attach (iocb_ptr, args, loud_sw, code);
```

1. iocb_ptr	is a pointer to an IOCB. (Input)
2. args	is an array of extent one, where args(1) is an attach description. (Input)
3. loud_sw	if ON, causes a message to be printed before returning a nonzero status code. (Input)
4. code	is a standard status code. (Output)

Internal Logic

A data block called t is allocated in the linkage section for each attachment. These blocks are threaded together and have the following format:

```
   dcl 1 t aligned based(table_ptr),
        2 attach_descrip char(12) varying,
        2 open_descrip char(27) varying,
        2 device_id char(6),  /* as in attach descrip */
        2 tty_index fixed bin,  /* from hcs_$tty_attach */
        2 el,
         3 no_channel fixed bin,  /* = 1 */
         3 event fixed bin(71),  /* channel id */
        2 flags aligned,
         3 vacant bit(1) unal, /* ON if block is free */
         3 tn1200 bit(1) unal, /* ON for TermiNet 1200 */
         3 pad bit(34) unal,
        2 next_block_ptr ptr,  /* forward thread */
        2 line_length fixed bin,  /* settable length */
        2 print_pos fixed bin(22),
        2 no_char fixed bin(22);
```

The list is searched for a vacant data block, either the first, which is always there, or a later one that was allocated and subsequently marked as free by a detach call. If there is no vacant block, one is allocated and threaded to the end of the list.

Common IPS interrupt masking strategy is used. Errors are handled by the internal procedure error_, which prints a message if loud_sw is on and aborts the attach operation by a nonlocal goto to a return statement.

An event channel must be assigned so that the hardcore I/O controller can block waiting for input from the terminal. A fast event channel, which lacks certain capabilities not needed by terminal I/O, is obtained by calling hcs_$assign_channel. If no fast channel is available, a full event channel is obtained by calling full_ipc_$create_ev_chn. The channel id is stored in the data block and passed to hcs_$tty_attach to perform the attachment. This last call returns an index, t.tty_index, used in further calls to hcs_ to perform I/O operations on the attached switch. If attachment is unsuccessful, the event channel is deleted and error_ is called.

The detach_iocb and open entry variables in the IOCB are set to the tty_detach and tty_open entry points in tty_.

The internal procedure set_mask is called by the entry points tty_detach, tty_open, and tty_close to implement common I/O system IPS interrupt masking strategy. In addition, this internal procedure sets the variable actual_iocbp to point to the actual IOCB for the attachment and sets the variable table_ptr to point to the data block for the attachment.

The internal procedure set_up, called by the entry points tty_get_chars, tty_get_line, tty_put_chars, tty_control, tty_modes, and tty_position, sets the two pointers but does not mask IPS interrupts because no masking is needed for these operations.

Entry: tty_$tty_detach

This entry point performs detachment. The IOCB is set to the detached state regardless of whether terminal detachment is successful.

Usage

```
     dcl tty_$tty_detach entry(ptr,fixed bin(35));

     call tty_$tty_detach (iocb_ptr, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. code   is a standard status code. (Output)

This entry point calls set_mask, deletes the event channel, and calls hcs_$tty_detach to perform the detachment. The IOCB is set to the detached state and iox_$propagate is called.

Entry: tty_$tty_open

This entry point opens a switch for stream input and/or output.

Usage

```
     dcl tty_$tty_open entry(ptr,fixed bin,bit(1),
                                             fixed bin(35));

     call tty_$tty_open (iocb_ptr, mode, extend, code);
```

1. iocb_ptr   is a pointer to an IOCB. (Input)

2. mode   specifies the opening mode:

   1 - stream_input
   2 - stream_output
   3 - stream_input_output

3. extend   is not used.

4. code   is a standard status code. (Output)

This entry point calls set_mask and sets the IOCB to the open state. The open description is determined from mode. If input is specified, the get_line, get_chars and position entry variables in the IOCB are set to the entry points tty_get_line, tty_get_chars, and tty_position If output is specified, the put_chars entry variable is set to tty_$tty_put_chars.

<u>Entry</u>: tty_$tty_close

<u>Usage</u>

```
dcl tty_$tty_close entry(ptr,fixed bin(35));

call tty_$tty_close (iocb_ptr, code);
```

1. iocb_ptr   is a pointer to an IOCB. (Input)

2. code   is a standard status code. (Output)

This entry point calls set_mask, sets the IOCB to the closed state, and calls iox_$propagate.

The entry points tty_$tty_get_chars and tty_$tty_get_line call the internal procedure read to read amounts of one line or less at a time. This procedure calls hcs_$tty_read. If an error occurs, the I/O operation is aborted by a nonlocal goto to a return statement. If no characters have been read, read goes blocked awaiting input from the terminal.

Entry: tty_$tty_get_chars

Usage

```
       dcl tty_$tty_get_chars entry(ptr,ptr,fixed bin(21),
                                fixed bin(21),fixed bin(35));

       call tty_$tty_get_chars (iocb_ptr, buf_ptr, buf_len,
                                amt_read, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. buf_ptr is a pointer to an input buffer. (Input)

3. buf_len is the number of characters to be read. (Input)

4. amt_read is the number of characters actually read. (Output)

5. code is a standard status code. (Output)

This entry point reads the specified number of characters by calling the read internal procedure repeatedly.

Entry: tty_$tty_get_line

Calling sequence is the same as for tty_$tty_get_chars.

This entry point calls the read internal procedure once. If the last character returned is not a newline character, code is set to error_table_$long_record.

Entry: tty_$tty_put_chars

Usage

```
       dcl tty_$tty_put_chars entry(ptr,ptr,fixed bin(21),
                                    fixed bin(35));

       call tty_$tty_put_chars (iocb_ptr, buf_ptr,
                                    buf_len, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. buf_ptr is a pointer to a buffer to be written out. (Input)

3. buf_len   is the number of characters to be written. (Input)

4. code   is a standard status code. (Output)

This entry point calls hcs_$tty_write. If an error occurs, the put_chars operation is aborted by a nonlocal goto to a return statement. If no characters have been written, the operation goes blocked waiting for previous output to be completed by the terminal.

<u>Entry</u>: tty_$tty_control

<u>Usage</u>

```
     dcl tty_$tty_control entry(ptr,char(*),ptr,
                                fixed bin(35));

     call tty_$tty_control (iocb_ptr, order,
                                info_ptr, code);
```

1. iocb_ptr   is a pointer to an IOCB. (Input)

2. order   is a character string specifying the control operation. (Input)

3. info_ptr   is a pointer to an info structure required by some operations. (Input)

4. code   is a standard status code. (Output)

The orders "resetread", "resetwrite", and "abort" are carried out by hcs_$tty_abort. All other orders are carried out by hcs_$tty_order.

<u>Entry</u>: tty_$tty_modes

<u>Usage</u>

```
     dcl tty_$tty_modes entry(ptr,char(*),char(*),
                                         fixed bin(35));

     call   tty_$tty_modes   (iocb_ptr,  new_modes,  old_modes,
code);
```

1. iocb_ptr   is a pointer to an IOCB. (Input)

2. new_modes is a character string specifying the intended modes. (Input)

3. old_modes is a character string specifying what the modes were. (Output)

4. code is a standard status code. (Output)

The modes operation is performed by hcs_$tty_order with the "modes" order.

<u>Entry</u>: tty_$tty_position

<u>Usage</u>

```
      dcl tty_$tty_position entry(ptr,char(*),
                              fixed bin(21),fixed bin(35));

      call tty_$tty_position (iocb_ptr, mode, records, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. mode is a null string. (Input)

3. records is the number of lines to space forward. (Input)

4. code is a standard status code. (Output)

The position operation is performed by repeated calls to the read internal procedure to read into a scratch buffer until records number of newline characters have been read.

SECTION VIII

DISCARD I/O MODULE

MODULE discard_attach.pl1

This module provides a sink for output to be discarded. The I/O operations put_chars, modes, write_record, control, and seek_key do nothing but return a zero status code. No other I/O operations except attach, detach, open, and close are supported.

Entry: discard_$discard_attach

Entry: discard_$discard_attach

Usage

```
      dcl discard_$discard_attach entry(ptr,(*)char(*),
                                     bit(1),fixed bin(35));

      call discard_$discard_attach (iocb_ptr, args,
                                     loud_sw, code);
```

1. iocb_ptr	is a pointer to an IOCB. (Input)
2. args	is an array of extent zero. (Input)
3. loud_sw	if ON, causes a message to be printed before returning a nonzero status code. (Input)
4. code	is a standard status code. (Output)

Internal Logic

Common IPS interrupt masking strategy is used. A data block called blk is allocated in the linkage section for each attachment. These data blocks are threaded together and have the following format:

```
   dcl 1 blk aligned based(blkptr),
         2 next ptr,  /* forward thread */
         2 attach char(8) varying,  /* attach descrip */
         2 open char(31) varying;  /* open descrip */
```

The attach description is discard_. The open description is null. The entry points discard_detach and discard_open in discard_ are filled into the IOCB.

Entry: discard_$discard_detach

Usage

```
dcl discard_$discard_detach entry(ptr,fixed bin(35));

call discard_$discard_detach (iocb_ptr, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. code is a standard status code. (Output)

Internal Logic

Common IPS interrupt masking strategy is used. If the detach_iocb entry variable in the IOCB is not equal to this entry point, the entry point in the IOCB is called instead. The IOCB is set to the detached state.

Entry: discard_$discard_open

Usage

```
        dcl discard_$discard_open entry(ptr,fixed bin,
                                        bit(1),fixed bin(35));

        call discard_$discard_open (iocb_ptr, mode,
                                        extend, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. mode is an opening mode. (Input)

3. extend is not used.

4. code is a standard status code. (Output)

Internal Logic

Common IPS interrupt masking strategy is used. If the open entry variable in the IOCB is not equal to this entry point, the entry point in the IOCB is called instead. The open description and the entry variables that are filled in depend on mode.

Entry: discard_$discard_close

Usage

```
        dcl discard_$discard_close entry(ptr,fixed bin(35));

        call discard_$discard_close (iocb_ptr, code);
```

1. iocb_ptr is a pointer to an IOCB. (Input)

2. code is a standard status code. (Output)

Internal Logic

Common IPS interrupt masking strategy is used. If the close entry variable in the IOCB is not equal to this entry point, the entry point in the IOCB is called instead. The entry points discard_detach and discard_open in discard_ are filled into the IOCB and the IOCB is set to the detached state.

SECTION IX

THE vfile_ I/O MODULE

INTRODUCTION

This I/O module implements the standard iox_ entry points for processing files in the Multics storage system. All logical file types and opening modes are supported.

PROGRAM MODULES

The vfile_ I/O module is composed of thirteen separate program modules. These programs and their associated data structures are described below.

The following programs are used with all file types:

vfile_attach.pl1
: implements attach, detach, open, and close entry points. Dispatches to the appropriate file-type module at opening and closing.

alloc_cb_file.pl1
: is used throughout vfile_ (and record_stream_) to maintain blocks of per-process storage as they are allocated and freed.

The following four modules are the main programs for each of the standard Multics file types. They implement all of the supported I/O operations except attach_iocb and detach_iocb. The procedure vfile_attach calls one of these programs at opening to set up the remaining IOCB entries and to allocate and initialize an IOCB. When a file is closed, vfile_attach calls a corresponding cleanup entry point in the appropriate module:

open_uns_file.pl1	for unstructured files
open_seq_file.pl1	for sequential files

open_blk_file.pl1   for blocked files

open_indx_file.pl1 for indexed files

The remaining program modules are called by open_indx_file to deal with indexed files:

find_key.alm
: is used to locate the first instance of a given key in the index, or the next larger key if not found.

change_index.pl1
: does the work of adding, deleting, and replacing keys in the index.

change_record_list.pl1
: manages assignment, reassignment, and freeing of blocks of record space.

create_seg_ptrs.pl1
: allocates and initializes, or frees, temporary storage for an array of pointers to the file's component segments. This module is called by change_index and change_record_list to obtain new multisegment file components.

create_position_stack.pl1
: allocates and initializes, or frees, a temporary array used to keep track of the current position in the index. The change_index module calls this program to increase the index tree height.

restart.pl1
: restores a file to a consistent state from any intermediate state that can exist as a result of an interrupted operation.

check_file_version.pl1
: transforms the headers of old version files into the new version.

INDEXED FILES

An indexed file is kept as a multisegment file with one or more index segments and zero or more distinct record segments. Each segment can contain up to 256K 36-bit words. Because the file is in the virtual memory, the implementation of vfile_ does not involve explicit I/O requests (I/O is done by the system's page control). However, all use of vfile_ is through a device-independent I/O interface with operations such as seek_key (locates a record), read_record, delete_record, etc.

Space for records is managed dynamically as records are written, rewritten, and deleted. A chained list of free blocks is kept, and allocation is by first fit with a roving pointer. Merging of adjacent free blocks is done with boundary tags (Knuth, vol. 1, p 442, Algorithm C). The space overhead is one word per allocated record. The minimum size of an allocated block is currently fixed at eight words. The end of a segment is treated specially so that the last nonzero word of the segment immediately follows the last allocated record.

To date we have no evidence that searching for a free block is a performance problem for anyone. (This is, of course, very dependent on the particular application.) However, as a result of some simulation studies, we are considering using a separate free list for each range of block sizes [2**m, 2**(m+1)-1], each list with its own roving pointer. This scheme is now used for PL/I areas in Multics and for general system dynamic allocation. Given a request for a block of size b, the first list searched is the one such that b is in [2**m, 2**(m+1)-1].

The index is kept as a B-tree. (R. Bayer and E. McCreight, Acta Informatica 1, pp. 173-189, 1972; and Knuth, vol. 3, Section 6.2.4). Each node occupies one page (1024 words). Keys are variable length $0 \leq \text{length} \leq 256$ characters (9-bit). For consistency with the PL/I (and Multics) rules for character string comparison, trailing blanks are ignored.

The layout of a node is as follows:

```
          ______Node (=1 page)_______
         |                           |
   ---|  last branch                |
     |   |___________________________|
     |   |                           |
  -+--|  last_key                   |
  ||     |___________________________|
  ||     |                           |
  ||     |  scattered_space (bytes)  |
  ||     |___________________________|
  ||     |                           |
  ||     |  branch ptr               |
  ||     |___________________________|
  ||     |                           |\
  ||     |  descriptor               | \
  ||     |___________________________|  one entry
  ||     |                           | /
  ||     |  branch ptr               |/
  ||     |___________________________|
  ||     |                           |
  ||     |  descriptor               |
  ||     |___________________________|
  ||     |                           |
  ||     |  .                        |
  ||     |  .                        |
  ||     |  .                        |
  ||     |___________________________|
  ||     |                           |
  ||->|  branch ptr               |
  |      |___________________________|
  |      |         ^                 |
  |      |         |                 |
  |      |  contiguous               |
  |      |  space                    |
  |      |         |                 |
  |      |         v                 |
  |      |___________________________|           Descriptor
  |-->|                           |      _________________________
         |    keys                   |  <---|key position|key length|
         |___________________________|      |_______record ptr______|
```

Descriptors are two words, each key is a string of from zero to 256 9-bit bytes, and the other items are one word each. The record and branch pointers are actually number pairs (component segment number in the file, offset in the segment). The key position and key length (each a half-word) locate the key string within the node. The variables last_branch and last_key together define the free space shown in the figure. The variable scattered_space gives the scattered free space available in the keys section (resulting from deletion of entries). The programs for insertion and deletion of an entry (branch, descriptor, and key) are roughly as follows:

Deletion

> Add size of entry to total available space.
> <u>If</u> total available space > 1/2 page, use underflow procedure.
> <u>Else</u> compact the array of branches/descriptors, setting last_branch = last_branch - 1, and add size of key to scattered space.

Insertion

> <u>If</u> size of entry <= contiguous free space, do a simple insertion.
> <u>Else</u> <u>if</u> size of entry <= total available space, compact the keys section and then do a simple insertion.
> <u>Else</u> use the overflow procedure.

The overflow procedure splits the node only if neither the left or right brother node has sufficient space available to correct the overflow by shifting some entries to the brother. The number of entries shifted is chosen to make the space used in each node as close to equal as possible. The underflow procedure is the usual one for B-trees. The node is balanced (by shifting entries) with its right brother if it has a right brother; otherwise the left brother is used.

Variable length keys introduce an effect not present with fixed length keys. Shifting entries between brother nodes (which also involves one entry in the parent) can cause the parent to overflow or underflow. Thus, it is possible for an underflow to cause the parent node to overflow. Fortunately, this possibility does not further complicate the program.

The I/O system distinguishes a special case of "keyed sequential output" for file creation or extension. For vfile_, this means that records are output in key order, i.e., are always appended to the file. In this mode, it does not shift entries when a node overflows. Instead, it splits with only one entry in the right half. This means that nodes on the right edge can contain only one entry, but all other nodes are proper and are almost full. Writing a file in the normal mode but with records in key order also results in very full nodes but takes much longer because of repeated shifts to balance the same pair of nodes.

SYNCHRONIZATION OF ACCESS

Introduction

The vfile_ I/O module supports concurrent retrievals and updates on indexed files. Any number of openings can exist simultaneously on a single file in any number of processes. This feature is optional, since it implies additional processing on each I/O operation to deal with possible asynchronous changes.

Synchronization is provided at both the file and individual record level through software locks. The use of these locks is to a large degree controlled automatically by vfile_. Additional synchronization can be achieved by explicit user control of locks and observance of certain protocols.

Data Structures

The file or index level synchronization involves the following permanent variables in the file header:

file_base.lock_word
: a standard Multics lock identifier used by set_lock_.

file_base.file_state_block.file_action
: a code indicating what operation, if any, is currently in progress.

file_base.change_count
: a counter whose value increases at the start of each file-altering operation.

Record level synchronization involves the following variables in each allocated record block:

record_block.lock_flag
: when set, indicates that the record is busy.

record_block.record_lock
: a standard Multics lock identifier located in the unused tail of the record block if space is available.

During a rewrite_record operation, the file_lock is treated as if it is a record_lock on the current record as well as a lock on the whole file.

Processing

I/O operations interact with the synchronization variables, depending on the use of the -share attach option and the class of operation.

In the following discussion, an operation is said to be "synchronized" with respect to a given lock if the operation references the lock (e.g., waits until it clears). Operations not inherently altering the file are termed "passive."

File Level

File level synchronization occurs at opening and on shared operations that reference an index component. One of two strategies is used depending on whether or not the operation is passive. The shared entries are distinguished from their unshared counterparts by a standard prologue and epilogue that surround the code for the body of the operation. When the -share option is not used, the only operations that reference the file lock are open and possibly close.

File-Altering Operations

The standard form of external entry points is:

```
operation: entry(args);
     indx_cb_ptr=iocb.open_data_ptr;
     if indx_cb.shared then call lock_file_check;
              .
              .
              .
     BODY OF OPERATION
              .
              .
              .
     go to unlock_exit;
```

where:

1. operation
   names the external entry point placed in the IOCB in normal openings (called by iox_$write_record, iox_$rewrite_record, etc.).

2. unlock_exit
   designates the block of code used as an external exit point for all nonpassive index operations. If not sharing, the program returns. Otherwise, the current index position (key and descriptor) and the file's change count are saved. The file is unlocked and the program returns.

3. lock_file_check
   is an internal procedure that begins by attempting to set the file's lock. Assuming that the lock can be set, the procedure sometimes reinitializes certain process variables to account for asynchronous changes (i.e., in other openings). If the file_action is nonzero, indicating an interrupted operation, the external procedure restart is called to complete and possibly undo the operation before proceeding.

The body of each such operation has the following general form:

1. Check args and abort if error.
2. Save necessary crash recovery information in the file's header.
3. Set file_action to indicate operation in progress.
4. Increment the file's change count.

5. Proceed with the file-modifying operation.

6. Set file_action to zero, indicating that the file is consistent.

<u>Passive Index Referencing Operations</u>

The standard form of external entry points for passive operations is:

```
     operation: entry(args);
          indx_cb_ptr=iocb.open_data_ptr;
          if indx_cb.shared then do;
               current_entry=n;
               go to init_entry;/*sets up handler*/
          retry_ent(n):
               call prepare_process;
          end;
               .
               .
               .
          BODY OF OPERATION
               .
               .
               .
          go to verify_done;
```

where:

1. operation
   names the external entry point placed in the IOCB in normal openings (called by iox_$read_record, iox_$read_length, etc.).

2. init_entry
   designates a block of code that saves initial values describing the state of the opening at the start of the operation (e.g., file position). The routine proceeds to establish an any_other handler and returns to retry_ent(current_entry). The handler transfers to retry_ent(current_entry) if the activating condition can have occurred as a result of asynchronous changes; otherwise the signal is passed on.

3. prepare_process
   is an internal procedure that saves the file's change count, waits for the file_action to become zero, and possibly reinitializes some process variables to account for asynchronous changes.

4. verify_done
   designates the block of code used as an external return point for all operations with passive index synchronization. If not sharing, the program returns. Otherwise, the current index position is saved and the current file change count is compared with the previously saved value. If no changes have occurred, the operation is "verified" and returns. Otherwise, the original file position information is restored and a transfer is made to retry_ent(current_entry), where the operation is reattempted.

Maintaining Correct Index Position

Each opening keeps track of its current index position between and during operations with the help of an array called the position stack.

In shared openings, the current key and record descriptor are saved at the end of each index-referencing operation. When other openings change the file's index, the information in the position stack becomes invalid. This situation is detected via the file's change count, and the saved key and descriptor are used to reseek the former position before the index entry is referenced.

The task of reseeking the saved position is assigned to the internal procedure restore_position. It is here that asynchronous insertions and deletions are detected in the case of shared operations. Note that the position can be restored to beginning or end of file as well as to a particular index entry, whichever is appropriate.

Proof of Passive File-Synchronization

The validity of the passive file synchronization strategy of vfile_ is guaranteed by the uniform file alteration protocol described earlier.

The diagram below represents a sequence of file alterations. Note that index-level alterations always occur serially, since each update begins by setting the file lock.

```
        A    B    C    D    E       A    B    C    D    E
<-------¦----¦----¦-XX-¦----¦-------¦----¦----¦-XX-¦----¦------>
```

The letters represent the following events:

A. Set file_lock (abort if busy).

B. Set file_action to nonzero code.

C. Increment change_count.

D. Set file_action to zero.

E. Unlock the file.

Each file transformation occurs between events C and D, marked XX above.

Passively synchronized references involve the following sequence of steps:

1. Save change count.

2. Wait for file_action to go to zero. (Abort if wait time exceeded.)

3. Compare change_count with saved count. If same then done, else go back to step 1.

The body of the passive reference takes place between steps 2 and 3.

What must be proven is that this sequence effectively excludes references during file transformations, i.e., within the intervals CD in the diagram above.

The classes of parallel references can be identified by the event, A-E, that immediately precedes step 1 of the reference. Consider each of the possible cases:

Step 1 follows event:

A. The reference must terminate before event C or else it will be repeated at step 3.

B. The reference will wait at step 2 and be repeated after event D.

C. The reference will wait at step 2 until D and terminate if step 3 occurs before the next event C. Otherwise, the reference is repeated.

D. The reference terminates if step 3 occurs before the next event C. Otherwise, it is repeated.

E. Same as D.

Thus, in no case can a reference terminate successfully if an intervening update (CD interval) occurs or has occurred.

Record Lock Processing

Record level synchronization occurs during operations that reference the length or contents of a record. As with file-level synchronization, one of two processing strategies is applied.

Record Locking and Unlocking

The following protocol applies to record alterations.

Explicit user operations -- record_status (set_lock="1"b):

1. The record's lock is set. (Abort if wait time exceeded.)

2. The record's lock_flag is set.

3. The change_count is incremented.

By user via pointer:

4. Record modification takes place.

record_status (clear_lock="1"b):

5. The lock_flag is cleared.

6. The record_lock is cleared.

Each record is thus constrained to be serially updated.

Passive Record Synchronization

A record is considered to be busy if its lock_flag is set. On each record reference, the lock_flag is tested and its status is indicated by the returned code.

There are two classes of passive record references to be considered. The first class includes the operations for which index synchronization also applies; the second class consists of references to records made directly via pointer (obtained with the "record_status" order).

The first class of operations requires no additional processing, since the index synchronization strategy detects all asynchronous changes by examining the file's change_count.

Users wishing to synchronize access to records via pointer without locking the entire file might consider a strategy like this:

1. Save change_count.

2. Check the record lock and continue if not busy.

3. Proceed with direct record reference.

4. Compare change_count with the saved count. Done if unchanged; otherwise must repeat operation.

A further refinement would be to introduce a user-supported change_count as the first word of each record. The record alteration protocol should be modified in this case to require that each rewrite increment the record's version number (change_count). Such a scheme might be justified when many processes are competing to perform frequent updates on (stationary) records in a single file.

RECOVERY FROM INTERRUPTIONS

Introduction

It can happen that while vfile_ is modifying a file, its execution is interrupted and not resumed (e.g., the system crashes). This can leave the file in a state where new operations cannot be performed, e.g., a node has been split but the new entry has not yet been made in its parent node. The program vfile_ has been coded so that the next time the file is used, the interrupted operation is automatically completed.

This feature requires the use of a substantial portion of each file header and a separate restart procedure. The rest of the mechanism is embedded throughout the file-altering sections of vfile_.

A uniform strategy applies, except in a few simple special-case situations. File-altering operations are designed to execute in either of two states, normal or repeating. In the normal state, each operation keeps track of its progress by saving certain variables in the file header. When an interruption is detected, the restart procedure reinvokes the interrupted operation in the repeating state. This results in the completion of the interrupted operation, whereupon the restart procedure returns, and the operation that detected the inconsistency proceeds normally.

The Normal State of Update Processing

The distinction between the normal and repeating states is made through the variable indx_cb.repeating. At opening, its value is set to "0"b, indicating the normal state.

On each file alteration, a certain amount of additional processing is done that is extraneous to the actual transformation that results. This extra work guarantees that any intermediate state of execution can be reconstructed and correctly resumed, provided only that the file itself is preserved intact.

For this purpose, two kinds of data are periodically saved in the file header during each update operation. First, there is the information that keeps track of the nature and degree of completion of an operation. Second, various external variables are saved that might otherwise perish with the user's process, or perish because of a subsequent assignment during the current operation.

Tracking Variables

In order to keep track of each operation's progress, the following variables are used:

file_base.file_state_block.file_action
: indicates which file-altering operation, if any, is currently in progress.

file_base.file_state_block.file_substate
: is a counter indicating how far the current operation has come toward completion.

file_base.index_state_block.index_action
: indicates which kind of index change, if any, is in progress.

file_base.index_state_block.index_substate
: is like file_substate, but applies only to the index alteration phase of the operation.

For each update operation, there is a corresponding file-action code that is set just before and cleared immediately after the file transformation takes place. Similarly, each index alteration is associated with a nonzero setting of index_action.

The substate counters are zeroed and periodically incremented during every transformation. By minimizing the frequency of substate changes, additional processing is reduced. This optimization, as it turns out, is largely achieved through otherwise arbitrary choices in coding style, such as the order of independent assignments.

Other Header Variables

The action and substate variables just discussed make up only a small part of the file header. Somewhat more than one page is reserved for the rest of the recovery-related file variables.

The remaining header variables are used during normal execution to save copies of certain other variables. Arguments and other external nonpermanent information that can affect the subsequent operation, e.g., file position, must be saved before any inconsistency is introduced. This precaution is required by the condition that the recovery mechanism always completes an interrupted operation. The other variables that must be duplicated are those permanent file variables that are altered subsequent to their affecting the course of the transformation.

Several optimizations apply to the saving of variables during updates. For example, the record argument is not saved during write and rewrite operations. This exception is handled by automatically deleting or flagging the record after restarting. Although it may be necessary to save many variables in a single update, the duration over which a given value must be saved is often shorter than the entire operation. Thus, a single header variable can serve as a repository for any number of separate values during the course of one operation. Another optimization that substantially reduces the cost of saving variables takes advantage of the efficiency of multiword assignments on Multics hardware.

The Restart Procedure

Whenever an entry point sets a file's lock, the header variable file_action is tested before proceeding with the body of the operation. If file_action is nonzero, an inconsistency exists in the file as the result of the interruption of a previous update operation. This situation is detected upon opening and at the start of shared update operations. It is dealt with simply by calling the external procedure restart.

The restart procedure performs the following simple tasks:

1. Saves the process information describing the state of the current opening (variables in the structure iocb.open_data_ptr->indx_cb).

2. Restores some arguments and process information for the interrupted operation, using values saved in the file_header.

3. Sets the variable indx_cb.repeating to "1"b and reinvokes the appropriate entry in open_indx_file to complete the interrupted operation.

4. Finally, after returning from the restarted operation, the process information for the current opening is restored and a return is made.

For the write_record, rewrite_record, and record_status operations, some additional steps are taken. In the case of rewrite_record, the user may be alerted to the potential inconsistency of the record's contents. For the other two operations, the new record is automatically deleted immediately after finishing the interrupted operation. This special treatment is required on writes and rewrites because efficiency considerations preclude saving the buffer argument at the start of every update.

The Repeating State of Execution

The last major feature of the recovery mechanism is the alternate state of update processing, characterized by the setting of indx_cb.repeating to "1"b. This situation only arises as a result of the detection of an interruption and invocation of the restart procedure discussed in the previous section.

What will ultimately be shown is that the result of reinvoking any interrupted operation in the repeating state is the same as it would have been, had the operation run to normal completion. Furthermore, the process of recovery must also be completely restartable.

To guarantee the correctness of restarting as described, it is sufficient to show that some set of conditions exist such that the total machine state (relevant to an operation) that existed just prior to any interruption is somehow reconstructed. The term "machine state" refers to both the state of execution (level of procedure invocation, for example) and the values of all variables that can subsequently be referenced. Since we are presumably dealing with a deterministic system, the replication of any prior state must produce the same outcome.

The essential difference between the two states of processing is that certain portions of code are bypassed in the repeating state. Otherwise, the flow of control is identical to that of normal execution. In restarting an operation, the repeating state automatically reverts to the normal state before reaching the point of interruption. Thus, the repeating state only applies to portions of code previously executed.

Sections of code to be skipped in the repeating state are embedded in internal procedures of the following form:

```
     (a "protected" procedure)
     routine_x:proc;
          if indx_cb.repeating then do;
               call check_file_substate;
               return;
          end;

          (body of procedure
           executed only in
           the normal state)
               .
               .
               .
          file_base.file_substate=
               file_base.file_substate+1;
     end routine_x;
```

where check_file_substate is the following procedure:

```
     check_file_substate:proc;
          indx_cb.next_substate=indx_cb.next_substate+1;
          if file_base.file_substate=indx_cb.next_substate
          then indx_cb.repeating="0"b;
     end check_file_substate;
```

Also, each update entry in open_indx_file starts with a call to the following internal procedure (some details omitted for clarity):

```
     initialize_substate:proc;
          if indx_cb.repeating
          then if file_base.file_substate=0
               then indx_cb.repeating="0"b;
               else indx_cb.next_substate=0;
          else file_base.file_substate=0;
     end initialize_substate;
```

Flow of Control

Half the problem of reconstructing the interrupted machine state is getting back to the right location in the code. If the program were completely linear, i.e., without any internal procedures or do loops, then a simple transfer would suffice. In general, the skipping mechanism used with the repeating state achieves the same end without the requirement of linear program flow. The correctness of this technique, however, does imply certain constraints.

To guarantee that flow of control returns to the point of interruption, it is required that the original path be followed, deviating only when it is certain that the bypassed code has already been completely executed, and in such cases always returning to the original path. Any control-altering statement that is repeated must therefore have the same outcome as before. This implies that any variables upon which a control-altering statement depends must be restored before the statement is repeated. Conversely, any control-altering statement that depends on a variable whose value can have changed must be skipped in the repeating state.

Reversion to the Normal State

The function of the internal procedure check_file_substate and the temporary counter indx_cb.next_substate is to ensure that the transition from repeating to normal execution takes place at the right moment. Strictly speaking, the right moment to stop skipping sections of normally executed code is the point after the last machine instruction executed before the interruption occurred. In general, however, some number of prior instructions can be repeated without altering the outcome. The permanent substate values delimit sections of code according to this property. Thus, for an interruption anywhere within a section of code corresponding to a single substate value, it is sufficient to revert to normal execution just prior to entering that section, or "logical block," of code.

The next_substate in the repeating state is initialized and periodically incremented so as to correspond to the normal substate value for the upcoming logical block. This practice allows the logical block of an interruption to be found simply by comparing next_substate with the permanent substate saved in the file_header. However, it should be noted that the mechanism for incrementing the next_substate described earlier introduces the constraint that such "protected" procedures not be nested. For this reason, a second permanent substate counter is used in the procedure change_index. Evidently, the use of multiple permanent substate counters effectively removes the constraint against nesting protected procedures.

Restoration of Variables

Having described the mechanism whereby flow of control returns to the point of interruption, it remains to be shown how the program variables are correctly restored to their previous values at the instant of reverting to normal execution. For this purpose, the variables are divided into two classes, distinguished by the constraints they impose upon protected procedures. All program variables upon which the completion of any update operation depends are required to fall into one of these classes.

Reconstructed Variables

A variable is "reconstructed" if every assignment to it is repeated and produces the same outcome as that prior to interruption. Thus a reconstructed variable cannot appear on the left of an assignment statement within a protected procedure. This definition guarantees that at any reference to such a variable while repeating, its value is the same as it was during previous normal execution. It follows, therefore, that when the reversion to the normal state takes place, all reconstructed variables have their former values, as required.

Protected Variables

A variable is "protected" if every assignment to it (except possibly the last) is skipped in the repeating state. Its value will therefore remain unchanged between the time an interruption occurs and normal execution is resumed. Protected variables must reside in the file, since only the file is assumed to be preserved.

A file variable can be protected first and then reconstructed, but not vice-versa. This constraint prevents any interrupted recovery from altering the protected value until it is no longer needed.

Statements that are repeated must have the same outcome in order to correctly reconstruct the interrupted machine state. This implies that no repeatable statement can depend upon any subsequently assigned protected variable.

The basis for subdividing the program into logical blocks, each corresponding to a substate value, lies in the dependencies on protected variables. Specifically, a single logical block is required to be independent of any protected variables subsequently altered in the same block. Otherwise, the outcome of reexecuting a block would depend on the point of interruption inside the block, which contradicts the defining assumption stated earlier.

Repeating State-Summary

Another point that was noted earlier is the requirement that the process of recovery from interruption itself be interruptible in the same sense. Fortunately, this problem has already been solved through the assumption that all variables are either reconstructed or protected. Since the file is thus constrained from changing its state until normal execution resumes, the only nontrivially distinct intermediate states are those associated with normal execution. Therefore an interrupted restart is always recoverable through the standard recovery mechanism.

SECTION X

DESIGN OF THE ANSI STANDARD AND IBM STANDARD TAPE I/O MODULES

INTRODUCTION

The tape_ansi_ I/O module implements the processing of magnetic tape files according to Draft Proposed Standard Revision X3L5/419T of American National Standard X3.27-1969, Magnetic Tape Labels and File Structure for Information Interchange. In addition, tape_ansi_ provides a number of features that are extensions to, but outside of, the specifications of the DPSR.

The tape_ibm_ I/O module implements the processing of both labeled and nonlabeled magnetic tape files in accordance with the standards specified in the following IBM publications: OS Data Management Services Guide, Release 21.7, GC26-3746-2; IBM System/360 Disk Operating System Data Management Concepts, GC24-3427-8; and, OS Tape Labels, Release 21, GC28-6680-4. The processing of nonlabeled tapes in DOS leading tape mark (LTM) format is not supported.

Both I/O modules operate in conjunction with IOX. As such, they are not called directly by the user but via iox_. The reader is urged to review the MPM documentation for the I/O modules for definitions of terms, summaries of standard specifications, and an overview of the collection of system I/O modules. Although they are documented as separate entities, both tape_ansi_ and tape_ibm_ are implemented in a single body of code referred to below as the I/O module.

Attach Function

The IOX attach function is performed by the external procedure tape_ansi_attach_. While the I/O module conforms to the IOX convention of attaching to a particular file, the nature of the storage medium requires specification in the context of membership in a logical file set residing on a physical volume set. The user can uniquely identify a particular file by supplying either the file identifier (the file's name) or file sequence number (within the file set), in combination with the volume name of the first volume of the volume set. The volume name of the first volume is also used as the external file set identifier. In addition, the attach description must specify all the information needed to perform the desired operation, i.e., processing mode, file attributes, number of devices to be used, etc.

The physical sequential organization of magnetic tape file sets implies that processing a file can affect the files following it. The internal logic of the I/O module must therefore regard the attach operation as attachment to a file set, and the processing of a particular file as a matter of positioning. In this light, an attachment may be categorized as either initial, if the attached file is a member of a file set that has never before been used (by the I/O module in the current process), or subsequent, if the file set has been used, even if the particular file specified has not. The former case causes a per-file set data base to be created in the process directory, while the latter relies on the information contained in such a data base.

The following major functions are performed at attach time:

1. The attach description is validated for self-consistency.
2. The per-file set data base is either located or created, and initialized.
3. The attach description is validated against the known file set characteristics, if any.
4. Volumes are mounted and/or demounted, as necessary.

Open Function

The open function is performed by the external procedure tape_ansi_file_cntl_ for ANSI and IBM SL file sets, or by tape_ansi_nl_file_cntl_ for IBM NL file sets. The opening modes supported are sequential_input and sequential_output. The I/O module does not support sequential_input_output because the sequential organization of magnetic tape file sets severely restricts the use of this opening mode. When opened for sequential_output, the I/O module operates in one of four output modes (extend, modify, generate, or create) specified at attach time. Not all output modes are supported for each type of file set organization.

The bulk of the I/O module's processing functions are performed at open time. Most of these are designed to ensure the validity of the I/O operations to be performed and the continued integrity of the file set. In addition, information on the structure of the file set as a whole is gradually added to the per-file set data base.

The following major functions are performed at open time:

1. The opening mode is validated against the attach description.

2. If the location of the desired file can be determined from the per-file set data base, the I/O module positions directly to that file. Otherwise, the I/O module searches the file set. In the course of searching, it adds information about the files it encounters to the file set data base. Volumes can be mounted and/or demounted, as necessary, as a part of the open function.

3. Once the desired file is located, its attributes are validated against those specified in the attach description.

4. The logical record I/O mechanism is initialized for the type of I/O operation to be performed.

Close Function

The iox_$close function is performed by the same external procedure that opened the I/O switch (tape_ansi_file_cntl_ or tape_ansi_nl_file_cntl_). The following major functions are performed at close time:

1. Logical record I/O is terminated in a consistent manner.

2. File processing is terminated in a manner that ensures the validity of the file and the integrity of the file set. The final state of the file is recorded in the file set data base.

Detach Function

The iox_$detach_iocb function is performed by the external procedure tape_ansi_detach_. The following major functions are performed at detach time:

1. Resource disposition is performed as specified by the attach description. If all volumes are demounted, the volume sequence list is purged of volume set candidates. The user is notified if the volume set membership appears to have changed..

2. If an inconsistency in the file set data base has been detected during the course of the attachment, all resources are released and the data base is deleted.

Other Functions

1. The iox_$read_record and iox_$write_record functions are performed either by the external procedure tape_ansi_lrec_io_, or by tape_ansi_ibm_lrec_io_, depending on whether the I/O module is processing an ANSI or IBM file set.

2. The iox_$control function is performed by the external procedure tape_ansi_control_.

3. The iox_$position function is performed by the external procedure tape_ansi_position_.

4. The iox_$read_length function is performed by the external procedure tape_ansi_read_length_.

5. No other iox_ functions are supported.

DATA STRUCTURES

Control Segment

For each file set accessed by the I/O module, a separate data base is maintained in the process directory. This data base is known as a control segment, or cseg. The cseg contains a variety of data describing both the logical and physical status of the file set. Some of these data are invariant, some change with each attachment, and some even change with each physical tape I/O operation. To minimize unnecessary processing each time an attachment is made, and to take advantage of data that may be accumulated during the course of several attachments, the cseg is maintained for the life of a process. If, however, an unresolvable inconsistency is detected in the cseg while processing an attachment, the cseg is deleted at detach time.

The cseg is divided into 4 major components: the cseg body, which contains data pertaining to the status of the file set as a whole; the physical I/O section, which contains data used by tape_ansi_tape_io_ and the tseg structure used by tdcm_; the volume chain, which describes the physical volumes that make up the volume set; and the file chain, which describes the logical files that make up the file set. The include file tape_ansi_cseg.incl.pl1 defines the cseg body, the physical I/O section, and the volume chain.

```
/*  BEGIN INCLUDE FILE:  tape_ansi_cseg.incl.pl1             */

dcl  cP ptr;

dcl 1 cseg based (cP),
    2 file_set_lock bit (1),
    2 invalid bit (1),
    2 standard fixed bin,
    2 attach_description,
      3 length fixed bin (17),
      3 string char (256),
    2 open_description,
      3 length fixed bin (17),
      3 string char (32),
    2 module char (12) varying,
    2 owner_id char (14),
    2 ndrives fixed bin,
    2 nactive fixed bin,
    2 write_ring bit (1),
    2 protect bit (1),
    2 density fixed bin,
    2 vcN fixed bin,
    2 fcP ptr,
    2 flP ptr,
2 hdw_status,
  3 bits bit (72) aligned,
  3 no_minor fixed bin,
  3 major fixed bin (35),
  3 minor (10) fixed bin (35),
2 lbl_buf char (80),
2 open_mode fixed bin,
2 close_rewind bit (1),
2 force bit (1),
2 user_labels bit (1),
2 no_labels bit (1),
2 output_mode fixed bin,
2 replace_id char (17),
2 retain fixed bin,
2 lrec,
  3 bufP ptr,
  3 nc_buf fixed bin,
  3 offset fixed bin,
  3 saveP ptr,
  3 file_lock bit (1),
  3 blkcnt fixed bin (35),
  3 reccnt fixed bin (35),
  3 code fixed bin (35),
2 read_length,
  3 rlP ptr,
  3 rlN fixed bin (21),
2 user_label_routine (6) variable entry (char (80), bit (1)),

2 syncP ptr,
2 mode fixed bin,
2 soft_status,
  3 nbuf fixed bin,
  3 buf (2),
    4 bufP ptr,
    4 count fixed bin,
2 (free_list, busy_list, chain (3), bufct (3)) fixed bin,
2 wait_switch (1:63) bit (1) unaligned,
2 tseg aligned,
  3 areap ptr,
  3 ev_chan fixed bin (71),
  3 write_sw fixed bin (1),
  3 sync fixed bin (1),
  3 get_size fixed bin (1),
  3 (ws_segno bit (18),
    drive_number fixed bin (17)) unal,
  3 buffer_offset fixed bin (12),
  3 buffer_count fixed bin (12),
  3 completion_status fixed bin (2),
  3 hardware_status bit (36) aligned,
  3 error_buffer fixed bin (12),
  3 command_count fixed bin (12),
  3 command_queue (10) fixed bin (6) aligned,
  3 bufferptr (12) fixed bin (18) aligned,
  3 buffer_size (12) fixed bin (18) aligned,
  3 mode (12) fixed bin (2) aligned,
      3 buffer (4) char (8192) aligned,

    2 vl (63),
      3 position,
        4 fflX fixed bin unal,
        4 cflX fixed bin unal,
        4 pos fixed bin unal,
        4 lflX fixed bin unal,
      3 vol_data,
        4 volname char (6),
        4 comment char (64) varying,
        4 rcp_id fixed bin (6),
        4 event_chan fixed bin (71),
        4 tape_drive char (8),
        4 write_VOL1 fixed bin,
        4 ioi_index fixed bin,
      3 reg_data,
        4 volume_id char (32),
        4 tracks fixed bin unal,
        4 density fixed bin unal,
        4 label_type fixed bin unal,
        4 usage_count fixed bin unal,
        4 read_errors fixed bin unal,
        4 write_errors fixed bin unal,

    2 chain_area area;

/*  END INCLUDE FILE:  tape_ansi_cseg.incl.pl1               */
```

cP	is a pointer to the cseg. It is set at attach time by a call to hcs_$make_seg, which either creates or initiates the cseg corresponding to a particular file set.
file_set_lock	is "1"b if the file set is currently in use, i.e., an I/O switch is attached to any file in the file set. It is set at attach time and reset at detach time.
invalid	is "1"b if an unresolvable inconsistency is detected in the cseg during the course of an attachment. It causes the cseg to be deleted at detach time.
standard	is the file set organization code: 1 - ANSI; 2 - IBM OS; 3 - IBM DOS. It is set when the cseg is created, and is invariant.
attach_description	is the iox_ attach description structure.
length	is the number of meaningful characters in the description string.

string	is the attach description string.
open_description	is the iox_ open description structure.
length	is the number of meaningful characters in the description string.
string	is the open description string.
module	is the name of the I/O module (either tape_ansi_ or tape_ibm_). It is set when the cseg is created, and is invariant.
owner_id	is the user's Person_id (or first 14 characters thereof), recorded in the VOL1 label if a volume must be initialized. It is set when the cseg is created, and is invariant.
ndrives	is the maximum number of devices that can be assigned in the course of an attachment. It is set at attach time.
nactive	is the actual number of devices currently assigned by the I/O module. It can be set at any time by tape_ansi_mount_control_.
write_ring	is "1"b when the volume set is (to be) mounted with write rings. It is set at attach time.
protect	is "1"b if hardware file protect is on; writing is inhibited, regardless of write rings. It is set at either attach or open time by tape_ansi_mount_cntl_.
density	is the density at which the file set is (to be) recorded. Zero indicates default, 2 indicates 800 bpi, and 3 indicates 1600 bpi. If specified, this field is set at attach time by tape_ansi_attach_. It may be (re)set at attach or open time by tape_ansi_mount_cntl_.
vcN	is the number of links (elements) in the volume chain (array) that actually contain volume data. The value is incremented for each volume added to the volume sequence list, and decremented for each volume set candidate purged at detach time.
fcP	points to the base of the file chain, the file data link, which is always allocated. It is set when the cseg is created.

flP	points to the file chain link currently (or last) in use. Its value can change during tne course of an attachment.
hdw_status	contains the hardware status data associated with every physical tape operation, as interpreted by tape_ansi_interpret_status_.
bits	is the IOM status string.
no_minor	is the number of minor status conditions associated with the major status.
major	is a standard status code indicating the major status condition.
minor	is an array of standard status codes indicating the minor status conditions.
lbl_buf	is the buffer into which volume and file labels are read, and from which they are written.
open_mode	is either 4 - sequential_input, or 5 - sequential_output. It is set at open time.
close_rewind	is "1"b if the volume currently in use when the I/O switch is closed is to be rewound. It can be set at any time by a close_rewind order.
force	is "1"b if unexpired files are to be overwritten without querying for permission. It is set at attach time if the -force option appears in the attach description.
user_labels	is "1"b if the user file labels are to be read or written. This feature is currently not supported.
no_labels	is "1"b if the file set does not contain volume or file labels; i.e., the file set organization is IBM NL. It is set at attach time if the -nlb option appears in the attach description.
output_mode	defines the type of output operation to be performed if the I/O switch is opened for sequential_output. Possible values are: 0 - no output permitted; 1 - extend existing file; 2 - modify existing file; 3 - generate existing file; 4 - create new file. This field is set at attach time.

replace_id	contains the file identifier of a file to be replaced as the result of a create-type output operation. It is set at attach time if the -replace option appears in the attach description.
retain	specifies the detach-time resource disposition. Currently, only three values are defined: 0 - unassign all devices and volumes, the default disposition; 1 - unassign all devices and volumes, explicit specification; 4 - retain all devices and volumes, explicit specification. It is set at attach time, and can be reset at any time by a retain_none or retain_all order call.
lrec	is used by tape_ansi_lrec_io_ or tape_ansi_ibm_lrec_io_ to control the blocking and deblocking of logical records. Its values are initialized at open time, and some are reinitialized for each file section processed.
bufP	points to the tseg buffer into which a block is read, or from which a block is written.
nc_buf	contains the number of characters in a buffer available for reading as logical records.
offset	contains the number of characters already extracted from a buffer in the read case, or the number of characters already placed into the buffer in the write case.
saveP	points to the last (or only) RDW in an IBM V, VB, VS, or VBS format block being constructed for output. If the block is padded to a multiple of 4 characters, the RDW must be modified to include the length of the padding. It is set by tape_ansi_ibm_lrec_io_.
file_lock	is "1"b if the file is currently in use. It is used to prevent conflicting operations on the file or the cseg from multiple command levels, and is set by tape_ansi_control_, tape_ansi_position_, tape_ansi_read_length_, and by the logical I/O procedures.
blkcnt	contains the number of physical blocks processed for each file section.
reccnt	contains the number of logical records processed for each file. It is not currently used.

code	if nonzero, indicates that an unrecoverable error has occurred and prohibits further I/O operations. The value is a standard status code. It is set by tape_ansi_control_, tape_ansi_position, tape_ansi_read_length_, and by the logical I/O procedures.
read_length	is used by tape_ansi_read_length_ and the logical I/O procedures to implement the iox_$read_length operation.
rlP	points to a segment in the process directory into which a logical record can be read. It is set by tape_ansi_read_length_.
rlN	contains the number of characters in the logical record. It is set by tape_ansi_read_length_ when a logical record is obtained form the logical I/O procedure in response to an iox_$read_length call. It is reset by the logical I/O procedure when the record is transmitted to the user in response to an iox_$read_record call.
user_label_routine	is an array of entry variables defining the user's label processing routines. The sequence of entries is: 1 - read UHL; 2 - write UHL; 3 - read UTL; 4 - write UTL; 5 - read UVL; 6 - write UVL. This feature is currently not supported.

Physical I/O Section

This section of the cseg is used only by tape_ansi_tape_io_, tdcm_, and in one case, by tape_ansi_mount_cntl_. When the I/O module is modified to use rcp_ for resource management and an ioi_ interface (tape_ioi_) for physical I/O, this section can be removed.

syncP	points to the tseg buffer reserved for synchronous I/O. It is set and used by tape_ansi_tape_io_.
mode	is zero if the hardware is to read/write in binary mode; 1 for 9 mode. It is set by tape_ansi_file_cntl_, and used by tape_ansi_tape_io_.
soft_status	contains the status of the tseg buffers after an unrecoverable write error has occurred while doing asynchronous I/O. It is set by tape_ansi_tape_io_ and is used by the logical I/O procedure to maintain a valid block count.

nbuf	contains the number of tseg buffers (blocks) not written.
bufP	points to the tseg buffer.
count	is the number of characters in the buffer.
free_list, busy_list, chain, bufcnt	are used by tape_ansi_tape_io_ to manage the 3 tseg buffers used for asynchronous I/O.
wait_switch	is an array of switches, one for each possible device, indicating whether or not that device is waiting for an ipc_ event to occur. The switches are used by tape_ansi_tape_io_ when a rewind order is issued, and by tape_ansi_mount_cntl_ when volumes are mounted or unloaded.
tseg	contains the data used by tdcm_ to perform physical I/O.

Volume Chain

The volume chain is a symbolic representation of those volumes that are members of the volume set, in the order in which they became members, followed by those volumes that are potential members, in the order in which they may become members. The number of active links is dynamically variable (specified by cseg.vcN), as is the point of demarcation between volume set members and volume set candidates.

vl	is a volume chain link. Volume links can be (re-)initialized at attach, open, or I/O time, as necessary.
position	describes the portion of the file set recorded on the volume and the current physical position of the volume in that context. These data are maintained by the open procedure.
fflX	is the first file link index, i.e., the index of the file chain link corresponding to the first file section recorded on the volume. If fflX = 0, no file set sections are recorded, and the volume is not a member of the volume set; it is, therefore, a volume set candidate.

cflX	is the current file link index, i.e., the index of the file chain link which corresponds to the file section at which the volume is positioned. If cflX = 0, the position of the volume is unknown.
pos	is the intrafile position code, defining the file section component at which the volume is positioned. Possible values are: 0 - in header label group; 1 - in data (passed header group tape mark); 2 - in trailer label group (passed data tape mark).
lflX	is the last file link index, i.e., the index of the file chain link corresponding to the last (or only) file section recorded on the volume.

vol_data — contains data used to perform volume mounting and labeling. These data are maintained by tape_ansi_mount_cntl_.

volname	is the volume name of the volume.
comment	is the message text displayed on the operator's console when the volume is mounted.
rcp_id	is currently the tdcm_ device index, required to perform all tdcm_ functions (both mounting, demounting, and I/O). If rcp_id = 0, the volume is not currently mounted. When rcp_ is used to perform resource management, it contains the rcp_ id code.
event_chan	contains the ipc_ event channel that has been created for use with the device on which the volume is mounted.
tape_drive	contains the name of the device on which the volume is mounted.
write_VOL1	governs the writing/validation of VOL1 labels. Possible values are: 0 - the first block is a valid VOL1 label and the volume identifier is correct; 1 - the tape is blank; 2 - the first block cannot be read; 3 - the first block is not a valid VOL1 label; 4 - the first block is a valid VOL1 label but the volume identifier is incorrect; 5 - the first block is a valid VOL1 label and the volume identifier is correct, but the density is incorrect.

ioi_index	is not currently used.
reg_data	will eventually contain data obtained from the tape registration file. With one exception, it is not currently used.
volume_id	contains the volume identifier (to be) recorded in the VOL1 label. Only six characters are used. This field is set by tape_ansi_mount_cntl_.

File Chain

The file chain is a symbolic representation of the files (file sections) that constitute the file set. Each file or file section is described by a file link, and the entire file chain is preceded by a file data structure that defines the mode of processing to be used for a particular attachment to a file. The file data structure is the first link of the chain. The file chain can vary in length dynamically but always contains at least one link, the file data structure.

chain_area	is the area in which the file chain is allocated. The chain area available in a 256K segment is sufficient to provide for approximately 3500 file sections.

File Data Structure (File Data Link)

```
/*  BEGIN INCLUDE FILE:  tape_ansi_fd.incl.pl1                  */

dcl 1 fd aligned based (cseg.fcP),
    2 backP ptr init (null),
    2 nextP ptr init (null),
    2 flX fixed bin init (0),
    2 vlX fixed bin init (0),
    2 dummy_HDR2 bit (1),
    2 eox fixed bin init (2),
    2 hdr1,
      3 file_id char (17),
      3 set_id char (6),
      3 dummy_section fixed bin,
      3 sequence fixed bin,
      3 dummy_generation fixed bin,
      3 dummy_version fixed bin,
      3 creation char (5),
      3 expiration char (5),
      3 access char (1),
      3 dummy_blkcnt fixed bin (35),
      3 system char (13),
    2 hdr2,
      3 format fixed bin,
      3 blklen fixed bin,
      3 reclen fixed bin (21),
      3 dummy_next_volname char (6),
      3 blocked bit (1),
      3 mode fixed bin,
      3 bo fixed bin,
      3 cc char (1);

/*  END INCLUDE FILE:  tape_ansi_fd.incl.pl1                  */
```

The file data link is structured identically to a (regular) file link. Some values are set when it is initially allocated, and are invariant. Other values are set on a per-attachment basis if certain options appear in the attach description. These values may be further set or reset at open time when the file is actually located and its labels examined.

fd	is the file data structure. It is allocated at the beginning of cseg.chain_area and is always pointed to by cseg.fcP.
backP	points to the previous file chain link, and is therefore always null.
nextP	points to the next file chain link, if any.
flX	is the file link index. Each file chain link has an associated index value, corresponding to the file section that it represents. The file links are assigned sequential index values. The index value assigned to the file data link is zero, because the file data link does not correspond to any file section.
vlX	is the volume link index. The file section represented by a file link is recorded on a particular volume, represented by a link in the volume chain. The volume link index thereby establishes the mapping between logical files and physical volumes. The value of vlX in the file data link is zero, because the file data link does not represent a file section.
dummy_HDR2	is not used.

eox	is always 2. This value means that the next file section must be recorded on a new volume. Since vlX for the file data link is zero, this eox value causes the first file section to be recorded on the first volume set member.
hdr1	contains data derived from, or to be recorded in, the HDR1 file label. HDR1 data describe the external characteristics of the file to be processed.
file_id	is the file identifier, or name, of the file to be processed.
set_id	is the file set identifier. The volume identifier of the first volume of the volume set is used.
dummy_section	is not used.
sequence	is the file sequence number, or position, of the file to be processed.
	dummy_generation, dummy_version are not used.
creation	is the date the cseg was created (today's date), in Julian form.
expiration	is the date on which the file to be created or generated will expire, if specified, in Julian form.
access	specifies access to the file. It is currently only used if an existing file contains access data in its file labels.
dummy_blkcnt	is not used.
system	is the system code that corresponds to files recorded by this I/O module. The system code for tape_ansi_ is "MULTICS ANSI "; for tape_ibm_, it is "MULTICS IBM ".
hdr2	contains data describing the internal structure attributes to be used when processing the file.
format	defines the logical record format. Possible values are: 0 - default, if determinable; 1 - undefined records, U format; 2 - fixed-length records, F/FB format; 3 - variable-length records, D/DB/V/VB format; 4 - spanned records, S/SB/VS/VBS format.

blklen	specifies the physical block length.
reclen	specifies the logical record length.
dummy_next_volname	is not used.
blocked	if "1"b, indicates that records are blocked.
mode	specifies the data encoding mode. Possible values are: 1 - ASCII, 9 mode; 2 - EBCDIC, 9 mode; 3 - binary.
bo	for ANSI files, specifies the block prefix length (the buffer offset). It is currently only used if an existing file was recorded with block prefixes.
cc	for IBM files, specifies whether or not records contain carriage control characters. It is currently only used if an existing file was recorded with carriage control characters.

File Link

There is a file link in the file chain for each file section known to exist. The number of file links can therefore increase in the course of processing. If an error occurs while creating a file section, the number of file links can decrease.

A special file link, the end-of-file-set-link (eofsl), is placed at the end of the file chain to follow the file link corresponding to the file section known to be the last file section in the file set.

```
/*   BEGIN INCLUDE FILE:  tape_ansi_fl.incl.pl1              */

dcl 1 fl aligned based (cseg.flP),
    2 backP ptr init (null),
    2 nextP ptr init (null),
    2 flX fixed bin init (0),
    2 vlX fixed bin init (0),
    2 HDR2 bit (1) init ("0"b),
    2 eox fixed bin init (0),
    2 hdr1,
      3 file_id char (17),
      3 set_id char (6),
      3 section fixed bin,
      3 sequence fixed bin,
      3 generation fixed bin,
      3 version fixed bin,
      3 creation char (5),
      3 expiration char (5),
      3 access char (1),
      3 blkcnt fixed bin (35),
      3 system char (13),
    2 hdr2,
      3 format fixed bin init (0),
      3 blklen fixed bin init (0),
      3 reclen fixed bin (21) init (0),
      3 next_volname char (6) init (""),
      3 blocked bit (1) init ("0"b),
      3 mode fixed bin init (0),
      3 bo fixed bin init (0),
      3 cc char (1) init (" ");

/*  END INCLUDE FILE:  tape_ansi_fl.incl.pl1                    */
```

File links are maintained by tape_ansi_file_cntl_. Some data are invariant for the life of the link. Others are set on a per-attachment basis from data obtained from the file data link in combination with the actual file labels. The file link describes the characteristics of a file section as it is (to be) recorded. The file data link describes the characteristics to be assumed when the file is processed. Whereas the two sets usually coincide, they need not.

fl	is a file link. It is based on cseg.flP, the file link pointer, which points to the link currently or last in use.
backP	points to the previous link in the chain.
nextP	points to the next link in the chain, if any. (nextP = null does not imply that the file section associated with the file link is the last of the file set.)
flX	is the file link index, the sequence of the file link within the file chain. For ANSI file sets, this value corresponds to the file section number. If flX = -1, this link is an eofsl. An eofsl does not correspond to any file section, but serves merely to indicate that the previous link is known to correspond to the last file section of the file set.
vlX	is the volume link index. The file section represented by this link is (to be) recorded on the volume link having this position in the volume chain.
HDR2	is "1"b if the file section is (to be) recorded with a HDR2 label.

eox		defines the state of trailer label processing for this file section. Possible values are: 0 - trailer labels have not been read/written; 1 - EOF labels have been read/written; 2 - EOV labels have been read/written. If eox = 2, the end of volume has been reached, therefore the next file section must be on the next volume of the volume set.
hdr1		contains values describing the external characteristics of the file. These data are obtained from, or are to be recorded in, the HDR1 label.
	file_id	is the file identifier, the name of the file.
	set_id	is the file set identifier, the volume identifier obtained from the VOL1 label of the first volume set member.
	section	is the file section number, the sequence of the file section within the file set.
	sequence	is the file number, the sequence of the file (of which this section is a component) within the file set.
	generation	is the file generation number.
	version	is the file generation version number.
	creation	is the date the file was created, in Julian form.
	expiration	is the date before which the file cannot be overwritten without user permission, in Julian form.
	access	contains the file access/security code.
	blkcnt	contains the block count (to be) recorded in the first trailer label (EOF1 or EOV1).
	system	contains the system code of the system that created the file section.

hdr2		contains data describing the internal structure of the file section. (Note that all sections of a file must have identical internal structure.) It is initialized with default values, some of which are invalid in certain contexts. Whether or not the information available in this substructure is actually recorded in a HDR2 label depends upon the value of fl.HDR2.
	format	specifies the logical record format. Possible values are the same as for fd.hdr2.format.
	blklen	specifies the physical block length.
	reclen	specifies the logical record length.
	next_volume	contains the volume name of the next volume in the volume set, if eox = 2 and the file section was created by the I/O module. It is recorded in the EOV2 label only.
	blocked	if "1"b, indicates that records are blocked.
	mode	specifies the data encoding mode. Possible values are the same as for fd.hdr2.mode.
	bo	for ANSI files, specifies the length of the block prefix.
	cc	for IBM files, specifies whether or not carriage control characters are recorded in each record.

SECTION XI

PROCEDURES IN THE ANSI STANDARD AND IBM STANDARD I/O MODULES

MODULE: tape_ansi_attach_

This module performs the iox_ attach function, and is invoked via a call to either iox_$attach_name or iox_$attach_ptr. It parses and validates the attach description, creates and/or initiates the control segment, and mounts or demounts volumes as necessary.

<u>Entry</u>: tape_ansi_attach_$tape_ansi_attach

When this entry point is called, the standard code in the cseg is set to 1, indicating an ANSI file set.

<u>Entry</u>: tape_ansi_attach_$tape_ibm_attach

When this entry point is called, the standard code in the cseg is set to 2, indicating (initially) an IBM file set. (If the -dos option appears in the attach description, the standard code will eventually be updated to 3, indicating an IBM DOS file set.)

<u>Usage</u>

```
     dcl (tape_ansi_$tape_ansi_attach,
          tape_ibm_$tape_ibm_attach) ext entry (pointer,
          (*) char (*) varying, bit (1) aligned, fixed bin (35));

     call tape_ansi_$tape_ansi_attach (iocbP, opt, com, code);
     call tape_ibm_$tape_ibm_attach (iocbP, opt, com, code);
```

The iox_ module determines the entry name of an I/O module's attach procedure according to the following convention:

module_name_$module_name_attach

The names tape_ansi_ and tape_ibm_ must therefore appear on the bound object segment (bound_tape_ansi_), and the entry names tape_ansi_attach and tape_ibm_attach must be retained.

1. iocbP	points to an IOCB. (Input)
2. opt	is the attach description, parsed by iox_ into an array of character strings. (Input)
3. com	if "1"b, permits the I/O module to call com_err_. (Input)
4. code	is a standard status code. (Output)

If the code returned is nonzero, the I/O switch is not attached. The following error_table_ codes can be returned:

bad_arg	invalid attach description.
noalloc	too many files in file set (>3500).
noarg	invalid attach description.
not_detached	I/O switch already attached.
file_busy	file (set) already in use.
inconsistent	invalid attach description.
invalid_cseg	invalid control segment - retry attachment.

In addition, any code returned by:

hcs_$make_seg	unable to create and/or initiate cseg.
tape_ansi_mount_cntl_	unable to mount a volume.
tape_ansi_parse_options_	invalid attach description.

Internal Logic

Each entry sets the standard code in the cseg appropriately, and transfers to the common body of code. The IOCB is checked to ensure that the I/O switch is not already attached; If the IOCB is already attached, an error exit is taken. The attach description is validated by a three-step process:

1. The internal procedure tao_init is invoked to initialize an attach option structure. This structure is an automatic storage version of the based structure defined by the include file tape_ansi_options.incl.pl1. Those structure elements that must be set with values derived from the attach description are initialized such that they are clearly unset; those elements that need not be set from the attach description are initialized to their default values.

2. The external procedure tape_ansi_parse_options_ (see below) is invoked to validate the attach option array and encode it into the attach option structure. Validation is confined to the criterion of self-consistency and is performed independent of a particular file set organization.

3. The internal procedure check_attopt (see below) is called to validate the attach options in context, i.e., ANSI, IBM SL, or IBM NL, and then forms an attach description string for later allocation in the cseg.

The above method of attach option validation permits the eventual use of a universal tape I/O module option parser: either tape_ansi_parse_options_ itself, or a replacement. If an error occurs during steps 2 or 3, an error exit is taken.

The external procedure hcs_$make_seg is called to create and/or initiate the control segment. The cseg resides in the process directory, and is named as follows:

module_name_XXXXXX_.cseg

where module_name_ is either tape_ansi_ or tape_ibm_, and XXXXXX is the normalized volume name of the first volume of the volume set. If the specified volume name is entirely numeric and less than 6 digits long, it has been normalized (by tape_ansi_parse_options_) by padding on the left with zeros to length 6. Nonnumeric volume names are 1 to 6 characters in length, with no leading or trailing blanks. If the cseg is initiated (i.e., it previously existed), then the attachment is considered as a subsequent (re-)attachment to the file set; if the cseg is created, the attachment is considered as initial.

INITIAL ATTACH

The cseg is initialized with certain per-process invariants, such as the I/O module name, login ID, etc., and with values to indicate the initial nature of the attachment. The internal procedure cseg_init is invoked to initialize the cseg with those per-attachment data derived from the attach options. The external procedure tape_ansi_tape_io_$attach (see below) is called to initialize the tseg portion of the cseg, as well as the I/O buffer management mechanism. The chain area at the end of the cseg is formatted as a PL/I area by calling the external procedure area_, and the file data structure is allocated. This structure forms the first link in the file chain, and contains values used to initialize the next link (which will be a true file link). The internal procedure vl_init is invoked to initialize a volume link for each volume specified in the attach description, and the first (or only) volume of the file set is mounted by calling the external procedure tape_ansi_mount_cntl_$mount (see below). Finally, the internal procedure fd_init is invoked to initialize the file data structure (the first link in the file chain) with per-attachment data derived from the attach options.

SUBSEQUENT ATTACH

The cseg file set lock is first checked to determine whether or not the file set is currently in use. If it is not in use, the cseg is locked; otherwise, an error exit is taken. If the file set density has been specified in the attachment description, and it differs from that used in the previous attachment, the volume set must not have been retained at detach time. If it has been retained, an error exit is taken. If the maximum number of devices to be used during the course of the attachment differs from that of the previous attachment, and the new number is less than the number of devices currently assigned, the excess number of devices must be unassigned.

To unassign these devices, the file chain is scanned to find the file link corresponding to the first section of the desired file. (If the file does not exist, the last link in the chain is used. If the chain does not exist, the index of the first volume in the volume set is used.) Having obtained an index into the volume chain from the located file link, the volume chain is scanned to locate those volumes currently mounted. The chain is first scanned upwards from the first volume to the target volume, demounting volumes and unassigning devices until the new maximum device limitation is satisfied. If the scan is completed without reaching this number, the volume chain is then scanned downwards, from the last volume to the target, until the limitation is met.

The target volume may not be mounted. An initial attachment must always mount the first volume of the volume set in order to obtain its volume identifier for use as the file set identifier. Subsequent attachments can, however, postpone mounting until open time, so that an attach-detach sequence without an intervening open does not incur unnecessary overhead.

The volume set specification obtained from the attach description is validated against the current contents of the volume chain. If the number of volumes specified exceeds the current number of volume links, new links are initialized by calling vl_init. Each volume whose position in the specification corresponds to an existing link must then be validated against the current value in the link. If the two do not match and the link volume is not a volume set member, the link volume is demounted (if necessary) and the link is reinitialized by calling vl_init. If the two do not match and the link volume is a volume set member, an error exit is taken. This determination permits volume set candidates to be replaced at any time, but precludes the insertion of a "foreign" volume into a valid volume set.

If the attach description requires that volumes have write rings but rings are not currently inserted, the external procedure tape_ansi_mount_cntl_$insert_rings is invoked to insert them. If hardware file protect is on, tape_ansi_mount_cntl_$write_permit is called to turn it off. If the attach description does not require write rings but write rings are currently inserted, and hardware file protect is off, tape_ansi_mount_cntl_$write_protect is called to turn it on.

Finally, cseg_init and fd_init are called to initialize the cseg and file data structures, respectively, with per-attachment data derived from the attach options.

COMMON TERMINATION

Both initial and subsequent attachments copy the attach description string generated by check_attopt into the cseg attach description structure, and set the string length variable. If an IBM DOS file set is being processed, the cseg standard code is updated to 3. Interrupts are then masked, the IOCB is modified to the indicated attachment, interrupts are unmasked, and control returns to the caller.

Internal Procedure

Entry: check_attopt

This procedure validates the attach options in the context of either ANSI, IBM SL, or IBM NL file set organization. The sequence of checking is:

1. Physical medium characteristics: number of tracks and density. These checks are common to all contexts.

2. Context-specific checks: mutual exclusiveness of ANSI and -dos options, etc. Each type has some particular combination of options that must be validated.

3. Output mode checks: mutual exclusiveness of -extend and -expires options, etc. Each output mode requires the presence or absence of other options.

4. Record format checks. Each logical record format places particular constraints upon the file attribute options.

Checks are performed in an order that minimizes superfluous processing.

MODULE: tape_ansi_file_cntl_

This module performs the iox_$open and iox_$close functions for ANSI and IBM SL file sets. In addition, it performs end-of-file, end-of-volume, and other miscellaneous file processing on behalf of the logical I/O procedure (tape_ansi_lrec_io_ or tape_ansi_ibm_lrec_io_).

Entry: tape_ansi_file_cntl_$open

This entry point performs the iox_$open function. It positions to the attached file, mounts and/or demounts volumes as needed, processes the file labels, and maintains the file and volume chains.

Usage

```
     dcl  tape_ansi_file_cntl_$open ext entry
          (ptr, fixed bin, bit (1) aligned, fixed bin (35));

     call tape_ansi_file_cntl_$open (iocbP, mode, mbz, code);
```

1. iocbP	points to the IOCB. (Input)
2. mode	is the opening mode. Possible values are: 4 - open for sequential_input; 5 - open for sequential_output. (Input)
3. mbz	must be "0"b. (Input)
4. code	is a standard status code. (Output)

If code is nonzero, an error has occurred and the IOCB is not open. The following is a nonexhaustive list of the error_table_ codes that can be returned:

duplicate_file_id	The requested opening for sequential_output would cause a file to be created whose file identifier already appears in the file.
file_aborted	A serious error occurred while writing file labels, and the defective file (section) has been successfully deleted from the file set.
file_busy	The file (set) is currently in use for other I/O activity.
incompatible_attach	The attach description does not permit the IOCB to be opened in the specified mode.
incompatible_encoding_mode	The specified data encoding mode conflicts with the other attributes of the file or file set.

incompatible_file_attribute	A specified file attribute conflicts with the actual structure of an existing file.
insufficient_open	Insufficient information regarding the file attributes has been supplied to open the file. The file does not have HDR2 labels.
invalid_block_length	The specified block length is invalid, or in conflict with the other file attributes.
invalid_cseg	There is an internal inconsistency in the control segment precluding further operations other than iox_$detach_iocb.
invalid_expiration	The specified expiration date is not equal to or earlier than that of the preceding file in the file set.
invalid_file_set_format	The file set format is not in accord with the applicable standard (ANSI or IBM).
invalid_label_format	A label format is not in accord with the applicable standard (ANSI or IBM).
invalid_record_length	The specified record length is invalid or in conflict with the other file attributes.
invalid_volume_sequence	The volume set membership has not been specified in correct switching sequence.
noalloc	The I/O module is unable to allocate any more storage for the file chain. The number of file sections exceeds 3500.
no_file	The file specified for reading, extending, modifying, generation, or replacement does not exist.
no_next_volume	Another volume is needed to continue processing, but no such volume is available.

unexpired_file	The specified output operation would overwrite a protected file. The operation was not performed.
unexpired_volume	The specified output operation would overwrite a protected volume. The operation was not performed.
uninitialized_volume	A volume was not (or could not be) initialized in a manner that would permit the specified operation.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. If either cseg.invalid or cseg.file_lock is "1"b, an error exit is taken. The consistency code (cc) is set to 0 and a cleanup handler is established. cc is an automatic variable used by consistent to determine the action required to maintain file set consistency. The value of cc is always in the range 0 - 2; an increase in value corresponds to an increase in needed function. The value 0 invalidates the volume position; 1 does the same and, in addition, truncates the file and volume chains; 2 does all the above and, if possible, truncates the file set. A handler for the area condition is established in case the file chain exceeds approximately 3500 links.

The variable search_id is set to the file identifier of the desired file. If the opening mode is sequential_input and -name XX was specified, search_id is set to XX; if -name XX was not specified, search_id is set to "". If the opening mode is sequential_output and -replace XX was specified, search_id is set to XX; if -replace was not specified, search_id is set to the identifier specified by -name XX. The value of search_id is used by desired_file to locate the file to be accessed.

The program searches the file chain for the desired file, beginning with the first file chain link (the file data structure, which is always allocated). If the desired link is not found, build1 is invoked to add a link to the chain and initialize it with data obtained from its logically associated file section.

If the link, either previously existing or just created, is an end-of-file-set-link (eofsl), the previous link describes the last file section of the file set. (Reaching the end of the file chain without encountering an eofsl implies that there are additional file sections, as yet unexamined.) If the link is an eofsl, append_file is called to determine whether or not the desired file can be created at the end of the file set; i.e., whether or not the attach and open descriptions jointly define such an appending. If they do, control is passed to the output section of the program; if they do not, the desired file does not exist, so an error exit is taken via valid_exit. This exit point does not invoke the consistency mechanism.

If the link is not an eofsl, desired_file is called to determine whether or not the file link being examined is that of the first section of the desired file. If it is not, and the link was just created by build1, build2 is invoked to complete the link initialization process, and control passes to build the next file link. If the link previously existed, control passes to determine whether or not the next link exists.

When the desired link is located, a number of checks are performed to ensure the correctness of the open operation. If the link existed previously and the opening is for any operation other than extension or modification, desired_check is invoked to validate the mapping between the logical file link and its associated physical file section. If the opening is for sequential_input, control passes to the input section of the program.

If the opening is for sequential_output, further checks are necessary. If the output mode is modify or extend, control is passed to extend_chain which extends the file chain out to the last (or only) section of the desired file. Chain extension is performed in a manner analagous to chain search, as described above. Every section of the file must have a link present/created in the chain, because the extend operation is performed on the last (or only) section of a file, and the modify operation requires data (the generation version number) found only in the trailer $ (and hence the file link) of the last file section. If the operation is generation or creation, and -expires <u>date</u> was specified, the expiration date must be valid against that of the preceding file (if any). Due to the physical sequential organization of tape files, an overwrite operation on any one file also destroys all those files (if any) which follow consecutively after it. To avoid the necessity of determining the expiration date of all such consecutive files, it is required that a file expire no later than its predecessor. Therefore, only the expiration date of the file actually accessed need be checked. If the specified expiration date would violate this protocol, an error exit is taken via valid_exit. If all checks have succeeded, control passes to the output section of the program.

Input

The procedure setup_for_read is invoked to fill in the file data structure with those attributes from the file link that are not supplied by the attach description. The procedure lrec_open is then called to perform final consistency checks on the file attributes, and to initialize the logical record I/O data structure, cseg.lrec. Finally, move_tape_ is called to position the volume set to the first data block of the first section of the desired file. If none of the above steps results in an error exit, the IOCB is set to the open state and the program returns.

Output

If the -force option was not specified and the output operation would overwrite an unexpired file, the write_permit entry of another_volume is called to query the user for permission. If permission is denied, an error exit is taken via valid_exit. If permission is granted, truncate_chains is invoked to free any file chain links beyond that of the desired file. Volume chain links of volumes following the one on which the desired file resides are also changed from membership to candidate status. The procedure build_eofsl is invoked to place an eofsl on the file chain immediately following the desired file link. The chains are truncated and "capped" with eofsl's because the output operation to be performed physically truncates the file set. Control then passes to one of the three output mode sections of the program.

Extend and Modify

In the extend case, the volume set is positioned to just beyond the last data block of the last section of the desired file. In the modify case, the volume set is positioned to just before the first data block of the first section of the desired file. The procedure setup_for_extend_modify is called to update the appropriate file link in accordance with the operation being performed. Any file attributes not supplied at attach time are obtained from the file link. The procedure lrec_open is called to perform final consistency checks on the file attributes and to initialize the logical record I/O structure, cseg.lrec. In the extend case, extend_check is invoked to ensure that the special requirements for consistent extension of FB files are met. If none of the above steps results in an error exit, the IOCB is opened and the program returns.

Generate and Create

Either setup_for_generate or setup_for_create is invoked to update the appropriate file link. Any file attributes not supplied at attach time are obtained from the file link. The procedure lrec_open is called to perform final consistency checks on the file attributes and initialize the logical record I/O structure. The procedure move_tape_ is invoked to position the volume set for writing (new) header labels, write_HDRs is called to actually write the HDR1 and HDR2 labels, and finally write_TM and back_TM are called to write the header label group's tape mark and to backspace over it. (This sequence provides for eventual user label processing.)

If none of the above steps results in an error exit, the IOCB is opened and the program returns.

Entry: tape_ansi_file_cntl_$data_eof

This entry point is called only by the logical record I/O procedure, when a tape mark is detected in the course of a read operation. It determines whether the EOF indicates end of file or end of file section. In the latter case, it switches volumes.

Usage

```
     dcl  tape_ansi_file_cntl_$data_eof ext entry
          (ptr, fixed bin (35));

     call tape_ansi_file_cntl_$data_eof (iocbP, code);
```

If code is nonzero, an error has occurred and processing should not continue. In addition to the error codes listed above, the following error_table_ code can be returned:

discrepant_block_count	The block count recorded in the file section's trailer labels does not agree with the block count maintained by the I/O module.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. It is not necessary to check either cseg.invalid or cseg.file_lock; the former must be "0"b, else the (calling) logical I/O procedure could not have been invoked, and the latter must be "1"b, because the logical I/O procedure has been invoked. The variable cc is set to 0. Cleanup and area condition handlers are established. If invoked, these handlers call consistent and then close the IOCB.

The intrafile position code in the volume link is immediately updated to indicate that the volume is now positioned in the trailer label group (by virtue of having read over a tape mark). The logical I/O procedure is then invoked at the $close entrypoint, to terminate the I/O for the section, synchronize the tape position, etc. If the file chain indicates that the trailer label group has never been processed, process_EOX is called. The block count obtained from the file labels is then compared with that maintained by the I/O module. If they differ, the error code error_table_$discrepant_block_count is returned. If the file section is the last (or only) section of the file, the status code error_table_$end_of_info is returned. This is not an error, but rather an indication that no more data exists in the file.

If the file section is not the last (or only) section of the file, the file link corresponding to the next section must be examined. If this link does not exist, build1 is invoked to create it. The link is then tested to determine if it is an eofsl. If it is, the required next file section is missing. In this case, the program calls consistent, closes the IOCB, and returns the error code error_table_$invalid_file_set_format. Otherwise, checks are made to ensure that the next file section is indeed the correct section, i.e., that it is of the same file and follows sequentially the section in which the EOF was detected. If either of these checks fails, the program exits as described for a missing section. Finally, move_tape_ is invoked to position to the first data block of the new section, cseg.blkcnt (the block count maintained by the logical I/O procedure on a per-file-section basis) is reset to 0, and the program returns.

Entry: tape_ansi_file_cntl_$data_eot

This entry point is called only by the logical record I/O procedure, when end of tape (EOT) is detected in the course of a write operation, and by tape_ansi_control_ in response to an "feov" operation to simulate the detection of EOT. It switches to the next volume of the volume set.

Usage

```
     dcl  tape_ansi_file_cntl_$data_eot ext entry
          (ptr, fixed bin (35));

     call tape_ansi_file_cntl_$data_eot (iocbP, code);
```

If code is neither zero nor error_table_$no_next_volume, an error has occurred and the IOCB is not open. The latter error code indicates that no additional volumes are available and further I/O activity should be prohibited.

The cseg pointer is obtained from the IOCB. For reasons described above, neither cseg.invalid nor cseg.file_lock need be checked. The variable cc is set to 2, and a cleanup handler is established to call consistent and close the IOCB. The variable close_eot is then set to "0"b to indicate that this EOT was not detected during the course of an iox_$close operation.

An area condition handler is established. The procedure next_volume is invoked to determine whether or not a volume switch can be performed. If one cannot, the program returns the status code error_table_$no_next_volume. This is not an error, but rather an indication that further I/O activity should be prohibited. If possible, the volume name of the next volume is saved in the file link of the current section. The procedure write_TM is called to write the end of data tape mark, write_EOVs is called to write an EOV trailer label group, write_TM is called again to write the two end-of-volume tape marks, and write_new_section is invoked to perform the volume switch. Finally, cseg.blkcnt is reset to zero for the new section's block count, and the program returns.

<u>Entry</u>: tape_ansi_file_cntl_$position_for_output

This entry point is called only by the logical record I/O procedure, before performing the first write operation. It rewrites the header label group tape mark and performs volume switching, if necessary.

<u>Usage</u>

```
dcl tape_ansi_file_cntl_$position_for_output
    ext entry (ptr, fixed bin (35));

call tape_ansi_file_cntl_$position_for_output
     (iocbP, code);
```

If code is neither zero nor error_table_$no_next_volume, an error has occurred and the IOCB is not open. The latter status code indicates that no additional volumes are available and further I/O activity is prohibited.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. For reasons described above, neither cseg.invalid nor cseg.file_lock need be checked. The variable cc is set to 2 and a cleanup handler is established to call consistent and close the IOCB. The procedure write_TM is invoked to rewrite the header label group tape mark. If EOT is detected when the tape mark is written, control passes to the data_eot section of the program, at the label eot_not_while_closing. This section operates as described above, causing a null file section to be written and volume switching to occur. The ANSI and IBM standards both require this action when EOT is detected in the header label group.

<u>Entry</u>: tape_ansi_file_cntl_$beginning_of_file

This entry point is called only by tape_ansi_position_ in response to a -1 (position to beginning of file) operation.

<u>Usage</u>

```
     dcl  tape_ansi_file_cntl_$beginning_of_file
          ext entry (ptr, fixed bin (35));

     call tape_ansi_file_cntl_$beginning_of_file
          (iocbP, code);
```

If code is nonzero, an error has occurred and the IOCB is not open.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. For reasons described above, neither cseg.invalid nor cseg.file_lock is checked. The variable cc is set to 0 and a cleanup handler is established to call consistent and close the IOCB. The file chain is scanned backwards from the current file link until the first (or only) section of the file is found. The procedure move_tape_ is then invoked to position the volume set to the first data block of the first section.

Entry: tape_ansi_file_cntl_$end_of_file

This entry point is called only by tape_ansi_position_ in response to a +1 (position to end of file) operation.

Usage

```
     dcl  tape_ansi_file_cntl_$end_of_file
          ext entry (ptr, fixed bin (35));

     call tape_ansi_file_cntl_$end_of_file
          (iocbP, code);
```

If code is nonzero, an error has occurred and the IOCB is not open.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. For reasons described above, neither cseg.file_lock nor cseg.invalid need be checked. The variable cc is set to 0. Cleanup and area handlers are established to call consistent and close the IOCB. The logical record I/O procedure is called at the $close entry point to terminate any I/O activity and to synchronize the tape. If the trailer labels of the current file section have not yet been processed, build2 is invoked. If the current file link corresponds to the last (or only) section of the file, move_tape_ is called to position the volume set to the first trailer label, and back_TM is called to position immediately following the last data block by backspacing over the end-of-data tape mark. The program then returns.

If the current file link does not correspond to the first (or only) section, the file chain must be searched (and perhaps extended) until the last section is located. If the next link in the chain does not exist, build1 is invoked to create it. If the next link is found to be an eofsl, the required next file section is missing. In this case, the program calls consistent, closes the IOCB, and returns the error code error_table_$invalid_file_set_format. Otherwise, checks are made to ensure that the next file section is indeed the next sequential section of the same file. If these checks fail, the program exits as described for a missing section. If the checks are successful, the program examines the link to see if it is the last section. The entire process is repeated until the last section is located.

Entry: tape_ansi_file_cntl_$close

This entry point performs the iox_$close function. In the read case, it merely terminates logical record I/O in a consistent manner and closes the IOCB. In the write case, it writes an EOF trailer label group (switching volumes if necessary) and closes the IOCB.

Usage

```
     dcl  tape_ansi_file_cntl_$close ext entry
          (ptr, fixed bin (35));

     call tape_ansi_file_cntl_$close (iocbP, code);
```

If code is nonzero, an error has occurred. In the write case, the file (or a portion thereof) can be invalid or destroyed. In any case, the IOCB is always closed.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. If cseg.file_lock is "1"b, the file is busy and an error exit is taken. Otherwise, the file is locked. The flag cseg.invalid is checked to ensure that the close mechanism can operate correctly. If the cseg is invalid, the IOCB is closed and an error exit is taken. A cleanup handler is established to close the IOCB in the same manner. If the IOCB was opened for sequential_input, an invalid cseg at close time is merely an inconvenient error. If the switch was opened for sequential_output, however, the consequences are far more serious. The file being closed is left in an inconsistent state, therefore the structure of the file set as a whole is inconsistent. This inconsistency is transparent unless an attempt is made to extend or modify the file, or to append yet another file to the file set. The inconsistency can be corrected by creating a new file in place of the defective one.

If the opening was for sequential_input, the variable cc is set to 0 and a cleanup handler is established to call consistent and close the IOCB. The logical I/O procedure is then invoked at the $close entry point. The current volume is rewound if a close_rewind order has been issued, and the IOCB is closed. The file is then unlocked (by setting cseg.file_lock to "0"b) and the program returns.

If the opening was for sequential_output, cc is set to 2 and a cleanup handler is established to call consistent and close the IOCB. Calling consistent (either due to cleanup or error) under these circumstances causes an attempt to delete the last (or only) section of the file. If the intrafile position code in the volume link associated with the current file section indicates that the volume is positioned within the header label group, no data records were ever written. In this case, write_TM is invoked to rewrite the header label group tape mark, in order to determine whether or not EOT has been reached.

If EOT is detected, control transfers to the data_eot section of the program (described above) at label eot_while_closing. A null file section is then recorded, volume switching occurs, and another null file section's header label group is recorded on the new volume. Control is then transferred back into the close section of the program at label continue_close to continue as though EOT had not occurred.

If EOT is not detected, the logical I/O procedure is called at the $close entrypoint. The end-of-data tape mark, EOF trailer label group, and two end-of-volume tape marks are then written. The volume is rewound if a close_rewind order was given, the IOCB is closed, the program unlocks the file and returns.

<u>Entries</u>: tape_ansi_file_cntl_$debug_on

tape_ansi_file_cntl_$debug_off

These entry points are used for debugging purposes. Most tape_ansi_file_cntl_ internal procedures print their names upon entry if a switch is "1"b. In addition, every file label read or written is printed (together with I/O status) if this switch is "1"b. Debugging output is directed to user_output.

<u>Usage</u>

```
tape_ansi_$debug_on   -or-  tape_ibm_$debug_on
tape_ansi_$debug_off  -or-  tape_ibm_$debug_off
```

These entry points are normally invoked from command level. They do nothing but set a static variable to either "0"b or "1"b, and can be invoked at any time, even before an IOCB is attached. Procedure trace and label printing are valuable tools for examining defective volumes, as well as for debugging tape_ansi_file_cntl_ itself.

Internal Procedures

In the following procedure descriptions, extensive use is made of the file chain concept. To review, the file chain consists of one or more structures (file links) allocated in cseg.chain_area. The links are interconnected by forward and backward pointers. The first link of the chain, the file data link, is always allocated. If the end of the file set has been empirically detected, a special link -- the end of file set link (eofsl) -- is added to the end of the chain. Each link in the chain, with the exception of the file data link and the eofsl, corresponds to a file section. In the paragraphs below, a clear distinction between the logical entity, the file link, and the physical entity, the file section, is not always maintained.

The base of the file chain is pointed to by cseg.fcP, whose value is constant for the life of the cseg. The file link currently being referenced is pointed to by cseg.flP, whose value is highly variable. Each link contains a file link index, fl.flX, and a volume link index, fl.vlX. File links are assigned successive indices beginning with the file data link, which is assigned index 0. The eofsl, if any, is assigned index -1. A positive index represents the absolute position of a file section within the file set. The index fl.vlX associates a volume link with each file link, the volume link corresponding to the volume set member on which the file section resides. The variable fl.vlX therefore represents the absolute position of a volume member within the volume set.

Entry: abort_file

This procedure is invoked by consistent when the consistency code (cc) is 2. It attempts to restore a valid file set structure by deleting the defective file section. If the defective section is the only section, the file itself is deleted. The external procedure command_query_ is invoked to obtain permission to delete the defective section. If permission is not granted, two tape marks are written and the entire file chain is invalidated. The tape marks can aid a subsequent retrieval effort using another tape reading mechanism. The procedure then exits with the error code error_table_$invalid_file_set_format.

If permission is granted to delete the defective section, the file link corresponding to the defective section is deleted and the file chain is "capped" with an eofsl. If the defective section is the first of the entire file set, initialize_volume is invoked to write a VOL1 label and dummy file. If the section is the first (or only) one of its file, but not of the file set, the volume set is positioned to just after the EOF trailer label group of the preceding file. If the section is not the first of its file, the volume set is positioned to just before the EOV label group of the preceding section, which is then rewritten as an EOF label group. Two tape marks are written, and the error code error_table_$file_aborted is returned.

If any of the above steps fails, a message is written on user_output stating that the file set structure is invalid. The entire file chain is then truncated, and the error code error_table_$invalid_file_set_format is returned.

<u>Entry</u>: another_volume

This function is invoked by next_volume when unable to determine the name of the next volume set member, if any. The external procedure command_query_ is invoked to determine whether the user wishes to terminate processing. If the answer is yes, the procedure returns the value "0"b. If the answer is no, command_query_ is again invoked to obtain the name of the next volume set member, along with an optional comment to be displayed at mount time. The procedure then returns the value "1"b, having placed the volume name in the global variable answer and the text of the comment, if any, in the global variable com_text.

<u>Entry</u>: write_permit

This function entry point is called from the output section of the $open code when the requested output operation would overwrite unexpired data. The external procedure command_query_ is invoked to obtain the user's permission to overwrite. If permission is granted, the function returns the value "1"b; if permission is denied, the function returns the value "0"b.

<u>Entry</u>: append_file

This function is called by the $open code when an eofsl is encountered while searching the file chain. It determines whether or not the attach-open combination can cause a file to be appended to the file set. A value of "1"b indicates that appending can be performed. If the attach-open does not specify sequential_output in -create mode, or if -replace was specified, appending is precluded and the procedure returns the value "0"b. If a -number option was not specified, a file sequence number is computed. If the option was specified, the sequence number must be one greater than that of the last file of the file set. If it is, the file can be appended; if not, the function returns the value "0"b. The expiration date of the file (either specified or defaulted) is checked to ensure that it does not exceed that of the previous file section (and by analogy, of the file set as a whole). If it does, a nonlocal exit is taken via the valid_exit mechanism, with the error code error_table_$invalid_expiration. Otherwise, make_eofsl_real is invoked to change the eofsl into a normal file link for the file about to be appended, and build_eofsl is called to cap the file chain with a new eofsl. The procedure then returns the value "1"b.

<u>Entry</u>: back_TM

This procedure backspaces over 1 or 2 tape marks, according to its calling arguments. The procedure tape_ansi_tape_io_$order is invoked to perform a "bsf" (backspace file) operation. If this call returns a nonzero status code, the procedure immediately returns that code to its caller. If the code is zero, the volume link intrafile and interfile position variables, vl.pos and vl.cflX, are adjusted to reflect the new volume position. If the procedure is requested to backspace more than 2 tape marks, this volume link mechanism does not work correctly. Having completed the requested operation(s), the procedure returns.

<u>Entry</u>: build1

This procedure adds a filled-in link to the file chain. build_fl is invoked to allocate storage for a new link and link it into the chain structure. The internal procedure move_tape_ is then called to position the volume set to the header labels of the file section corresponding to the new link. The internal procedure read_HDR1 is called to read the HDR1 label. If a tape mark is read instead of a HDR1 label, the end of the file set has been reached. The link index (fl.flX) is therefore set to -1, making it an eofsl, and reference to it is deleted from its associated volume link. The procedure then returns.

If a HDR1 label was read, fill_fl_from_HDR1 is invoked to validate and store the HDR1 information into the link. The internal procedure read_HDR2 is called to read the HDR2 label, if any. If this label is present, fill_fl_from_HDR2 is called to validate and store its information into the link. The procedure then returns. If any of the above calls returns a nonzero status code, the procedure immediately returns that code to its caller. The value of cseg.flP is set to the newly added link.

Entry: build2

This procedure completes a file link by inserting information obtained from the trailer label group of the associated file section. The procedure move_tape_ is invoked to position the volume set to just before the trailer label group. The procedure process_EOX is then called to read the labels and fill in the file link. If either of the above calls returns a nonzero status code, this procedure immediately returns that code to its caller. The value of cseg.flP is unchanged.

Entry: build_eofsl

This procedure appends an eofsl to the file chain, indicating that the end of the file set has been encountered. Storage for the link is allocated in the cseg.chain_area, and fl.nextP is set to point to the newly allocated storage. (The variable cseg.flP must point to the file link associated with what is, or will become, the last section of the file set.) The eofsl's backward chain pointer and link index are set. The value of cseg.flP is not changed.

Entry: build_fl

This procedure is called by build1 to add an empty link to the file chain. Storage for the link is allocated in cseg.chain_area, and fl.nextP is set to point to the newly allocated link. The variable cseg.flP must point to the last link in the file chain. The new link's backward chain pointer is set, and cseg.flP is updated to make the new link the current link. Control then passes to the section of the procedure described below, under the make_eofsl_real entry point.

Entry: make_eofsl_real

This entry point is used to change an eofsl into a normal link. The method used is identical to the initialization portion of the new link creation process.

The variable fl.flX is set to one greater than that of the previous link, as all file chain links must have sequential indices. If the previous link corresponds to an initial or medial section of a multisection file, the current link must reside on a different volume, the next sequential volume set member. The volume link of this volume is therefore set to indicate that the file section associated with the link is (will be) both the first and last section on the volume. If the previous file link corresponds to the last (or only) section of a file, the new link's volume index is set to the same value as that of the preceding link, and vl.lflX (the last file link index of the associated volume link) is incremented by one, indicating that another file section exists (or will exist) on the volume. The procedure then returns with cseg.flP pointing to the new (current) link.

Entry: consistent

This procedure is invoked when tape_ansi_file_cntl_ is unable to complete an operation without an error. The action performed is contingent upon the value of cc, the consistency code. If cc = 0, the position of the current volume is invalidated and the procedure returns. This is the usual case. If cc = 1, the volume position is invalidated and truncate_chains is invoked to delete the current, and subsequent, file links. This action is taken when an error occurs while modifying the file chain. If cc = 2, abort_file is invoked to perform the above operations in addition to physically truncating the file set. This action is taken when an error occurs while modifying the file set itself.

Entry: creating_first

This function is invoked to determine whether or not an attach-open combination specifies the creation of the first file of a file set. The criteria for a positive determination, indicated by a return value of "1"b, are:

1. The opening mode is sequential_output.
2. The output mode is -create.
3. The file sequence number, either specified or computed, is 1.
4. The -replace option was not specified.

If any of the above criteria are not met, the procedure returns the value "0"b.

Entry: desired_check

This procedure is called by the $open section of the program to validate file chain and volume chain data produced by a previous invocation of tape_ansi_file_cntl_. If a discrepancy between the file/volume chain and the file/volume set is detected, one attempt is made to resolve the discrepancy. (For example, an operator inadvertently rewinding a volume could cause a discrepancy between the "remembered" and actual volume position, but this discrepancy could be resolved by repositioning.) If it cannot be resolved, the cseg is invalidated and a nonlocal transfer is made to er_exit with the error code error_table_$invalid_cseg.

The procedure move_tape_ is invoked to position the volume set to just before the header label group of the desired file section. The procedure read_HDR1 is then used to read the HDR1 label. Detection of a tape mark at this point invokes the discrepancy mechanism. If the file set is ANSI, the file identifier and section number from the HDR1 label are compared with those values stored in the file link. If the file set is IBM, only the file identifier is compared. If the data differ, the discrepancy mechanism is invoked; if they agree, the procedure simply returns. If either of the above calls results in a nonzero status code, the procedure immediately returns that code to its caller.

Entry: desired_file

This function is called by the $open section of the program when the file chain is being searched and/or built. The procedure is invoked on each link in the chain to determine whether or not that link corresponds to the desired file. A positive determination is indicated by the return value "1"b. If the link does not correspond to an initial file section, the procedure immediately returns the value "0"b. (There is no need to examine medial/final links of a multisection file once it has been determined that the initial link does not correspond to the desired file.)

Three variables are used to make the determination: the file sequence number, the file identifier specified by the -name option, and the file identifier specified by the -replace option. A fourth variable, search_id, is set to a file_identifier value determined according to the opening mode. It is search_id that actually specifies the file identifier of the desired file, if known. From one to all of the first three variables can have values, in any combination. As the number of possible valid combinations is large, the algorithm used to determine desired file status can best be understood by referring to the code.

If the determination is positive, this procedure sets the file sequence number in the file data structure (in case it was not specified) and returns the value "1"b. If the determination is negative, the procedure checks to be sure that the attach-open combination does not require the duplication of a file identifier within the file set. If duplication is indicated, a nonlocal transfer is made to valid_exit with the error code error_table_$duplicate_file_id. Otherwise, the procedure returns the value "0"b.

Entry: extend_check

This procedure is called by the extend_file code in the $open section of the program. It determines whether or not the last block of a file being extended must be rewritten, and if necessary, manipulates the cseg so that the necessary operations will be performed at either first data write or close time.

A rewrite operation is necessary only when the last block of an FB format file contains fewer than the maximum possible number of records. If the file is found to be in FB format and to contain at least one block, tape_ansi_tape_io_$order is called to position the volume back over the last (or only) data block. (The extend_file code has already caused the volume to be positioned immediately after the last block, the cseg.lrec structure to be initialized, etc.) The entry point tape_ansi_tape_io_$sync_read is called to synchronously read the last block into the I/O buffer reserved exclusively for synchronous I/O calls.

Although the algorithm used to determine whether or not the block must be rewritten differs for ANSI and IBM (principally due to ANSI block padding conventions), the test is essentially two-fold. If the block does not contain an integral number of records (a defect since records should be fixed-length), or if another record cannot fit in the block, the procedure simply returns. Otherwise, tape_ansi_tape_io_$order is invoked again to backspace over the block. The entry point tape_ansi_tape_io_$get_buffer is called to assign an asynchronous I/O buffer, and the block is copied into it from the synchronous buffer. The procedure then returns.

The rationale behind the last steps runs as follows. When the logical record I/O procedure is called to write the first data record, it will find an asynchronous I/O buffer already assigned. It therefore places records into that buffer, effectively adding them to the last block, until the block is full. It then causes the buffer to be written. But the volume is positioned immediately before the last block, so that the write operation effectively rewrites a "full" last block. If the IOCB is closed without performing an intervening write operation, the fact that an asynchronous buffer is assigned at close time causes that buffer to be written. In this case, the rewritten last block is identical to the original. If any of the calls described in the above paragraphs returns a nonzero status code, the procedure immediately makes a nonlocal transfer to er_exit. In no case is the file itself modified at open time.

Entry: fill_XXX1

This procedure is called by all three entries in the write_HDRs procedure (write_HDRs, write_EOFs, and write_EOVs). It formats the first label of a file label group as either a HDR1, EOF1, or EOV1 label. As all nonconstant values are obtained from the file link, they have been derived from existing file labels and/or specified attach option values. As the format of ANSI and IBM XXX1 labels are virtually identical, referencing only the ANSI label structure suffices for both.

Entry: fill_XXX2

This procedure is called by the same three entries as fill_XXX1. It formats the second label of a file label group as either a HDR2, EOF2, or EOV2 label. As all nonconstant values are obtained from the file link, they have been derived from existing file labels and/or specified attach option values. As the format of ANSI and IBM XXX2 labels differ markedly, separate sections of code are necessary.

<u>Entry</u>: fill_fl_from_HDR1

This procedure is called by build1. Once a HDR1 label has been read (by read_HDR1), fill_fl_from_HDR1 is invoked to validate the label contents and encode them into the file link. The validation process can cause dynamic volume initialization, in the case of creating the first file of a new file set on a volume that does not begin with the first file of an old file set.

Because of the differences between ANSI and IBM labels, separate sections of code are required. Upon entry, a handler for the conversion condition is established. Most errors in label format can be detected simply by converting from strings to integers. Control is then passed to either the ANSI or IBM portion of the procedure.

The ANSI portion initially copies (and converts) a number of HDR1 fields into the file link. Checks are then performed to establish volume sequence validity. If the file section number is 1, and the file sequence number is 1, then the file section is the first of the entire file set and its file link should be the first of the file chain; otherwise, an error exit is taken. If the sequence number is not 1, there are two possibilities. Either the file link is the first of the chain, or it is not (indicating that this is not the first volume processed). If it is not, then the previous file link must correspond to a file with an EOV label set. If the link indicates an EOF label set, the current volume cannot possibly be part of the volume set, and an error exit is taken. If the file link is the first of the chain (indicating that this is the first volume processed), a problem can exist. A file whose sequence number is not 1 cannot be the first file on the first volume, unless a new file set is about to be created. The procedure creating_first is invoked to make this determination. If the determination is negative, an error exit is taken; otherwise, the volume is reinitialized, its HDR1 label reread, and the entire ANSI HDR1 processing code is reexecuted.

If the file section number is not 1 and the file link is the first of the file chain, the new file set creation checking described above is employed. Otherwise, the section number found in the previous file link is checked to ensure that sections are processed in correct ascending order. Finally, the volume sequence checking having been completed, the remainder of the HDR1 label data is encoded into the file link.

In the IBM case, the HDR1 label actually contains a volume sequence indicator, but does not contain a file section field. If the volume sequence field is 0, then the HDR1 label must be a standard dummy HDR1 label which appears at the beginning of a newly-initialized volume; otherwise, an error exit is taken. If the HDR1 label is correct and the file link is first in the chain (indicating that the volume is the first processed), the only valid case is that of creating the first file of a new file set. creating_first is invoked to make this determination. If the determination is negative, an error exit is taken. If none of the above tests fail, the file link is filled and the procedure returns.

If the volume sequence number is nonzero, the HDR1 label should be complete. The section number in the file link is set, to 1 if the file link is the first of the chain, otherwise to one greater than the section number of the previous link. A series of checks are then performed to ensure that the volume sequence number, file link index, and file sequence numbers are consistent, both among themselves and in relation to the previous file link. If all tests succeed, the remainder of the file link is filled in and the procedure exits.

<u>Entry</u>: fill_fl_from_HDR2

This procedure is called by build1 if read_HDR2 succeeded in reading a HDR2 label. It performs some simple validity checks, and encodes the HDR2 field values into the file link. As the ANSI and IBM HDR2 labels differ considerably, two separate sections of code are used.

Upon entry, an on unit is established for the conversion condition. Most label format errors can be detected in this manner. Control then passes to either the ANSI or IBM portions of the procedure.

In the ANSI case, the standard HDR2 fields are first encoded into the file link. If the system field in the HDR1 label is nonblank, the HDR2 buffer offset field is encoded. If the system field matches the I/O Module's system code, furthermore, the system-specific HDR2 fields are also encoded into the file link.

In the IBM case, the HDR2 format and block length fields are encoded and the density field is validated against the volume set density. If the dataset_position field is zero, the file must reside on the first volume. The file link's volume link index (fl.vlX) must therefore equal one. If the dataset_position field is nonzero, the file section cannot reside on the first volume. The index fl.vlX must therefore not equal one, and the previous file link must indicate a file section with an EOV label set. The remainder of the HDR2 fields are validated and encoded.

Entry: fill_fdhdr2_from_fl

This procedure is called by setup_for_extend_modify and setup_for_generate to validate the user-specified file data against the HDR2 label data, if any, in the file link. The file data values are overridden by their file link counterparts.

Entry: fill_flhdr2_from_fd

This procedure is called by setup_for_create, setup_for_extend_modify, and setup_for_generate to provide all the file link HDR2 data from the file data structure prior to an output operation. In the case of file creation, default values are determined for those HDR2 data not found in the file data structure. These defaults are applied to both the file data and the file link. In all other output cases, the lack of a file data value causes a nonlocal transfer to er_exit with the error code error_table_$insufficient_open.

Entry: file_new_section_fl

This procedure is called by write_new_section to initialize the file link of an about-to-be-written new file section. Most data are merely copied from the previous file link.

Entry: handler

This procedure is called by the any_other on units that are enabled, either at close or open termination, prior to IOCB manipulation. Any faults occurring while the IOCB is in an inconsistent state must be handled by this procedure.

The variable mask is checked to determine whether or not IPS interrupts have been masked. If they are, the external procedure terminate_process_ is called to terminate the user's process, because the IOCB manipulation cannot be completed. If they are not, the IOCB is still valid, and the external procedure continue_to_signal_ is called to propagate the condition.

Entry: initialize_permit

This function is called by move_tape_ to query the user for permission to initialize a volume. It returns a bit indicating whether or not permission is granted. (The standards provide for automatic initialization of blank tapes and correctly labeled expired volumes.) Based upon the value of vl.write_VOL1, the VOL1 label valid/invalid indicator, control is passed to one of an array of labels. Each section initializes the particular query to be issued and transfers to the common code.

Entry: initialize_permitA

This function entry point is called by fill_fl_from_HDR1 and write_new_section to query the user for permission to reinitialize an unexpired volume. It returns a bit indicating whether or not permission is granted. (The expired/unexpired status of the first file section on a volume is sufficient to determine the status of the entire volume.) After initializing the query to be issued, control passes to the common code.

The common code completes the initialization of the query_info structure and calls the external procedure command_query_. If the returned answer is "yes" (permission granted), the procedure returns the value "1"b. Otherwise, it returns the value "0"b.

Entry: initialize_volume

This procedure is called by abort_file, fill_fl_from_HDR1, move_tape_, and write_new_section to initialize a volume. Volume initialization consists of writing a VOL1 label and, depending upon the standard, one or two file label - double tape mark sequences.

The volume's vl.cflX is invalidated and tape_ansi_tape_io_$order is invoked to rewind the volume. An ANSI or IBM VOL1 label is constructed in cseg.lbl_buf A state variable is set to indicate that either two (ANSI) or one (IBM) file label - double tape mark sequences are to be written. The procedure write_label is called to write the VOL1 label.

An ANSI or IBM dummy HDR1 label is placed in cseg.lbl_buf, and write_label is called to write it out. The procedure tape_ansi_tape_io_ is invoked to write two tape marks. In the ANSI case, the same process is repeated for an EOF1 label - double tape mark sequence.

The result is that the volume is initialized according to the standard:

```
     ANSI           VOL1 HDR1 * * EOF1 * *

     IBM            VOL1 HDR1 * *
```

where * represents a tape mark.

An ANSI volume is initialized with a valid first file structure, but that an IBM volume is not. If an error occurs during any step of the initialization process, the procedure returns whatever error code it received from write_label or tape_ansi_tape_io_$order.

<u>Entry</u>: lrec_open

This procedure is called by the $open code to perform final validation of the logical record characteristics and to initialize the cseg for file opening. Validation consists of checking the record and block lengths in their contextual setting (i.e., depending upon file set standard, opening mode, and record format). If either the record length or block length is invalid, a nonlocal transfer is made to er_exit with error code error_table_$invalid_record_length or error_table_$invalid_block_length, respectively. In the IBM case, the encoding mode is also validated and if it is binary, a similar transfer is made with the error code error_table_$invalid_encoding_mode.

The cseg is initialized. The procedure tape_ansi_tape_io_$open is invoked to initialize the tseg buffer management strategy. The variable cseg.lrec.blkcnt is set to zero in the input case and to fl.blkcnt in the output case. In all output modes but extend, fl.blkcnt contains zero. In extend mode, it contains the current block count. This convention ensures that the block count eventually written in the EOF1 or EOV1 label (taken from cseg.lrec.blkcnt) reflects the cumulative block count resulting from an extend operation.

<u>Entry</u>: move_to_EOD

This procedure is called by the extend_file portion of the $open code to position the volume set to immediately beyond the last data block of the file. The file chain is scanned starting with the current link (corresponding to the first file section) until the last file section link is found. (The extend_chain portion of the $open code has already built the file chain to that last link, if it did not already exist.) The procedure move_tape_ is called to position the volume set to the trailer labels of that section, and back_TM is called to backspace over the end-of-data tape mark. These steps correctly position the volume set for file extension. If either of the above called procedures returns an error code, a nonlocal transfer is made to er_exit.

<u>Entry</u>: move_tape_

This procedure is responsible for volume set positioning and implements the I/O Module's triadic (volume index, file section index, intrasection position) position specification mechanism. It is the only internal procedure that itself contains internal procedures, and is therefore documented in the style of an external procedure.

<u>Usage</u> call move_tape_ (vX, fX, posit, ecode);

1. vX	is the index of the volume link corresponding to the desired volume. (Input)
2. fX	is the index of the file link corresponding to the desired file section. (Output)
3. posit	is the intrafile section position code: 0 - HDR1 label 1 - first data block 2 - EOF1 or EOV1 label (Input)
4. ecode	is a standard status code. (Output)

The variable vl.rcp_id is checked to determine whether or not the desired volume is currently mounted. If it is not mounted and the number of assigned drives (cseg.nactive) equals or exceeds the user-specified maximum (cseg.ndrives), the internal procedure find_candidate is invoked to determine the volume index of a volume that can be demounted. The procedure tape_ansi_mount_cntl_$remount is then called to demount the volume located by find_candidate and mount the desired volume on its drive. If the number of assigned drives is less than the user-specified maximum, tape_ansi_mount_cntl_$mount is invoked to assign a new device and mount the desired volume. Should this request for new device assignment cause the process device limit to be exceeded, move_tape_ recovers automatically by performing the find_candidate/tape_ansi_mount_cntl_$remount sequence described above.

Whether or not the desired volume was mounted when move_tape_ was invoked, it is mounted at this point. The volume's drive number (vl.rcp_id) and IPC event channel id (vl.event_chan) are placed in the tdcm_ tseg portion of the cseg. This step makes known, independent of the volume link index, the parameters that determine the device currently being used by the I/O Module. When tape_ansi_tape_io_ is replaced by an rcp_/tape_ioi_ interface, the tseg will be obviated; however, the current device parameters should still be maintained outside of the volume link, in order to provide for device I/O independent of the file chain/volume chain mechanism.

The status of the VOL1 volume label is determined by checking vl.write_VOL1, which was set by tape_ansi_mount_cntl_ when the volume was mounted. This variable specifies whether or not a VOL1 label need be written, and if so, why. A nonzero value indicates that a new label should be written. If the opening mode (cseg.open_mode) is for sequential_input, the need to write a label is an unrecoverable error; i.e., the tape's VOL1 label characteristics preclude its being processed as specified by the attach description. In this case, the external procedure ioa_ is invoked to print an explanatory message <u>via</u> user_output, normally directed to the user's terminal. The message text varies with the value of vl.write_VOL1, explicitly specifying the discrepancy detected between the attach description and the actual volume characteristics. The procedure then exits with the error code error_table_$uninitialized_volume.

If, however, the opening mode is for sequential_output, the volume can be (re-)initialized, contingent upon the successful completion of a series of checks. If vX, the volume link index parameter, specifies the first volume of the volume set, then the function creating_first is invoked. If the attach description specifies creation of other than the first file of the file set, the operation is invalid because (re-)initializing the first volume set member effectively truncates the file set. An explanatory message is issued as described above, and the procedure exits with the error code error_table_$uninitialized_volume. If vl.write_VOL1 indicates that the tape is blank, no further checks need be performed; otherwise, the procedure initialize_permit is invoked to query the user for permission to initialize. If permission is denied, the procedure exits with the error code error_table_$uninitialized_volume. If permission is granted, the procedure initialize_volume is called to initialize the volume. If a nonzero error code is returned, the procedure exits with that code; otherwise, vl.write_VOL1 is set to 0, indicating that the volume no longer requires (re-)initialization.

The procedure begins to position the tape. The automatic variable can_retry, initialized to "0"b upon procedure activation, is set to "1"b. When an error is detected within move_tape_, control always passes to the error exit code labeled error. This code invalidates the volume position (vl.cflX) and tests can_retry. If it is "1"b, it is set to "0"b and control passes to the label retry, which restarts the positioning operation. If it is already "0"b, the procedure returns with whatever error code is set. This algorithm provides one opportunity to resynchronize a volume's position with its volume link position data.

If the volume position is unknown or in the VOL/UVL label set (vl.cflX = 0), the internal procedure move_to_first_HDR is invoked to position the volume to the first HDR1 label on the tape. (This procedure call is labeled retry.) Once this has been done, vl.cflX is set to the index of the first file section on the volume (vl.fflX). The intrafile position indicator (vl.pos) is also known, and is set to indicate the HDR label group.

If fX, the file link index parameter, is greater than vl.cflX, move_tape_ must position forward a calculated number of tape marks. Positioning is done by the internal procedure move_forward. If fX is less than vl.cflX, the internal procedure move_backward is invoked, and if fX is equal to vl.cflX, either move_forward or move_backward is invoked depending upon the value of vl.pos. Even if vl.pos is equal to posit, the intrafile position parameter, move_backward is invoked to ensure that the tape is positioned at the initial block of the desired position and not at an indeterminate medial point. vl.cflX and vl.pos are then set to fX and posit, respectively, indicating that the requested positioning operation has been successfully performed, and the procedure returns. Should an error occur during any of the above steps, a transfer to error is made with whatever error code has been detected.

Internal Procedures

<u>Entry</u>: find_candidate

This procedure searches the volume chain for a mounted volume to be demounted, allowing the desired volume to be mounted in its place. The search goes from the first volume set member to the volume preceding the desired volume, and then from the last volume chain entry to the volume link following the desired volume. The first mounted volume ends the search. The algorithm results in minimum mounting/demounting if volume processing is performed in the usual manner, i.e., sequentially, from first to last volume set member. If no mounted volume is found, the procedure performs a nonlocal transfer to error with the code error_table_$invalid_cseg. This is done because find_candidate is never invoked unless move_tape_ has determined that a candidate for demounting does exist, by comparing cseg.ndrives with cseg.nactive.

<u>Entry</u>: move_to_first_HDR

This is the procedure that actually implements tape positioning by issuing calls to tape_ansi_tape_io_$order. It contains the entry points move_forward and move_backward, as well as the entries move_to_first_UHL and move_to_first_UTL, neither of which is currently used.

The entry point tape_io_$order is called to rewind the volume. The procedure read_label is invoked to read a label, and the call is repeated until a label beginning with HDR is encountered. (This call is labeled HDR_search.) The entry point tape_io_$order is called again to backspace to the beginning of the label, and the procedure returns.

<u>Entry</u>: move_forward

This entry point calls tape_ansi_tape_io_$order to forward space over as many tape marks as is necessary to perform the desired positioning operation.

<u>Entry</u>: move_backward

This entry calls tape_ansi_tape_io_$order to perform all but one of the backspace file operations necessary to correctly position the tape. The last such operation is also performed by tape_ansi_tape_io_$order, but the error code is specially checked to determine whether or not the volume was left positioned at beginning of tape. If it was, control transfers to HDR_search to effect positioning to the HDR1 label, as opposed to the VOL1 label. Otherwise, tape_ansi_tape_io_$order is invoked to forward space over the last tape mark encountered, leaving the tape correctly positioned.

Entry: next_volume

This function is called by process_EOX, as well as by the mainline entry data_eot, to determine whether or not the next volume set member already exists or can be created. A return value of "1"b indicates that it can.

If the volume link index of the current file link (fl.vlX) is less than the highest volume chain index (cseg.vcN), the procedure immediately returns "1"b because the next volume is already known. If fl.vlX is 64, the implementation restriction on the maximum number of volumes, then the external procedure ioa_ is called to issue an explanatory message and the procedure returns "0"b. This case is not treated as an error in the usual sense, because file set processing can and must be continued as though no other volume were available.

Having derived no information from the volume chain, the procedure examines the current file link to determine whether or not the name of the next volume can be extracted from the trailer label set. If it cannot, another_volume is invoked to query the user for the name of the next volume, if any. Should both of the above fail to provide the next volume, the procedure returns "0"b. If either does, cseg.vcN is incremented to reflect the addition of a new volume link and vl_init is called to initialize it. The volume name is then set in the new volume link, and the procedure returns "1"b. The algorithm ensures that a volume name entered into the volume chain (from the attach description, etc.) can override the one specified in an EOV2 label, if the one field exists.

Entry: process_EOX

This procedure is called by build2, as well as by the mainline entry data_eof, to read the trailer label set, validate its contents, and store information in the file link.

An on unit for the conversion condition is first established that transfers control to an error exit, bad_EOX, which returns the error code invalid_label_format. The procedure read_label is invoked to read the first trailer label. If this read encounters a tape mark, the procedure returns the error code error_table_$invalid_file_set_format because either an EOF1 or EOV1 label must be present. The file link trailer type (fl.eox) is set according to whether the label read is an EOF1 or an EOV1. This variable indicates whether or not volume switching is to be performed upon detection of a tape mark (indicating end of file section) while reading data, i.e., whether or not the file section is the last (or only) section of the file. If the label is neither EOF1 nor EOV1, the procedure returns the error code error_table_$invalid_file_set_format.

Data, such as generation version number and block count, are extracted and stored in the file link. The procedure read_label is invoked again to read the second trailer label, if any. If the read operation encounters a tape mark, no EOF2 or EOV2 label exists; tape_ansi_tape_io_$order is therefore called to space back into the trailer label set. If the first label was an EOF1, processing is complete and the procedure returns. If the first label was an EOV1, another file section must reside on the next volume. The procedure next_volume is called to determine its name, if possible. The volume name, if found, is stored in the file link and the procedure returns. Otherwise, the procedure returns the error code error_table_$no_next_volume.

If a second trailer label is present, it must be either a UTL or of the same type as the first trailer label. An UTL is treated as though no second label were read at all. If the types differ (e.g., EOF1 and EOV2), the procedure returns the error code error_table_$invalid_file_set_format. If the types match and the trailer set is EOF, the EOF2 label need not be processed (as it is an exact duplicate of the HDR2) and the procedure returns.

If the second label is an EOV2, indicating an ANSI file set, and the label contains the name of the next volume, that name is stored in the file link and the procedure returns. If any of the above are not satisfied, next_volume is invoked to attempt to determine the name of the next volume. The name, if one can be determined, is stored in the file link and the procedure returns. Otherwise, the error code error_table_$no_next_volume is returned.

<u>Entry</u>: read_HDR1

This procedure is called by build1, desired_check, and fill_fl_from_HDR1 to read and validate a HDR1 label. The parameter eofsw is first set to "0"b. This variable indicates whether a tape mark, as opposed to a HDR1 label, is read. The procedure read_label is invoked to read a label into cseg.lbl_buf. If a label is successfully read, it must be a HDR1 label. Otherwise, the error code error_table_$invalid_file_set_format is returned. If a tape mark is detected, the volume link intrafile position indicator (vl.pos) is incremented to reflect the tape mark crossing. The procedure back_TM is called to backspace over the tape mark, the eofsw parameter is set to "1"b, and the procedure returns. A tape mark where a HDR1 label would otherwise be found indicates the logical end of the volume. In addition, since read_HDR1 is never invoked once an EOV trailer set has been processed on a volume, finding a tape mark also indicates the logical end of the file set. If an error is detected while reading or backspacing, the procedure returns that error code.

Entry: read_HDR2

This procedure is called by build1 to read and validate a HDR2 label. The file link variable fl.HDR2 is first set to "0"b. This variable indicates whether or not the file section contains HDR2 (and, by implication, EOF2 or EOV2) labels, and therefore whether or not the file attributes (block length, record format, etc.) can be obtained from the file itself. The procedure read_label is invoked to read a label into cseg.lbl_buf. If a HDR2 label is read, fl.HDR2 is set to "1"b. Any other label (i.e., UHL label) is ignored. If a tape mark is detected, vl.pos is incremented and back_TM is called to backspace over it. The lack of a HDR2 label is not an error.

Entry: read_label

This procedure is called by move_tape_, process_EOX, read_HDR1, and read_HDR2 to read a file or volume label into cseg.lbl_buf. The entry point tape_ansi_tape_io_$sync_read is called to synchronously read one tape block. If a block is successfully read, its length is checked to determine whether or not the block could be a label. A length of less than 80 characters causes the procedure to return the error code error_table_$invalid_label_format. If the block length is 80 characters or more, it is moved from the synchronous I/O buffer to cseg.lbl_buf. If the file set is IBM standard, the external procedure ebcdic_to_ascii_ is called to perform character code conversion.

Entry: write_label

This entry point is called by initialize_volume and write_HDRs to write a file or volume label from cseg.lbl_buf. The label is first moved from cseg.lbl_buf into the synchronous I/O buffer. If the file set is IBM standard, the external procedure ascii_to_ebcdic_ is called to perform character code conversion. The entry point tape_ansi_tape_io_$sync_write is then invoked to synchronously write the label.

<u>Entry</u>: setup_for_create

This procedure is called from the $open code to initialize a file link preparatory to file creation. Since the file link describes a new entity, it must be completely filled in from user-specified data, invariable creation-specific values, and (if necessary) a set of file attribute defaults. The HDR1 portion of the file link data is initialized in part from the file data structure and in part from constants within the procedure itself. The procedure fill_flhdr2_from_fd is then invoked to complete the HDR2 portion of the file link.

<u>Entry</u>: setup_for_extend_modify

This procedure is called from the $open code to initialize the file data structure preparatory to file extension or modification, and to modify and/or complete the file link. The file name and sequence number are set in file data from fl.file_id and fl.sequence, respectively, in case one or the other was not explicitly specified in the attach description. If the file set is ANSI standard, the file link version number is incremented to reflect the pending operation. The file link creation date is set to the current date. If the operation is file modification, the current file block count (fl.blkcnt) is zeroed, because modification truncates the file. Otherwise, the block count is left as is, because the operation of file extension leaves the file's current contents unaltered and the block count must therefore be incremented from its initial value.

The procedure fill_fdhdr2_from_fl is invoked to fill the HDR2 portion of the file data structure from the file link. This step ensures that any file attributes specified in the attach description do, in fact, match the attributes recorded in the HDR2 label, if any. The procedure fill_flhdr2_from_fd is invoked to fill the HDR2 portion of the file link from the file data structure, <u>without</u> applying any defaults. This step ensures that the pending operation is not performed unless a complete, consistent attribute set has been composed from the attach description, the HDR2 label (if any), or both.

Entry: setup_for_read

This procedure is called from the $open code to initialize or complete a file data structure preparatory to reading a file. The file link is never changed to conform to the file data, and the two can in fact differ as regards the file attributes. A file can therefore be processed according to a more or less arbitrary set of file data (user-specified) attributes, while preserving the file's actual characteristics in the file link.

The file identifier and sequence number are first copied from the file link into the file data structure, in case one or the other was not specified in the attach description. If the record format was user-specified (fd.format ^= 0), the record b ing attribute (fd.blocked) is also known. If they are not specified, the record format and blocking attribute must be obtained from the HDR2 portion of the file link. The block length (fd.blklen) and record length (fd.reclen) are similarly checked and, if necessary, their values are set from the file link. In the case of the record length, however, defaults can in some cases be applied when the file link contains no information. Under all other circumstances, failure of the file link to supply a record format, record length, or block length not specified in the attach description results in a nonlocal transfer to er_exit, with the error code error_table_$insufficient_open.

If the file set is ANSI, the block prefix length (fd.bo) is set from the file link because this value, if present, is invariant. If the file was written by the I/O Module, the blocking attribute and character encoding mode (fd.mode) are also set from the file link, if not user-specified. If neither user-specified nor obtainable from the file link, they are set to the ANSI defaults, blocked and ASCII, respectively.

For an IBM file set, the blocking attribute and encoding mode are set to blocked and EBCDIC, respectively, if not user-specified.

Entry: setup_for_generate

This procedure is called from the $open code to initialize the file data structure preparatory to file generation, and to modify and/or complete the file link. The file name and sequence number are set in the file data from the file link, in case one or the other was not specified at attach time. The file link generation number (fl.generation) is incremented by 1, modulo 10000. (The largest possible value is 9999.) The generation version number is set to 0, indicating a new generation, and the creation and expiration dates (fl.creation and fl.expiration) are set from the file data. The procedure fill_fdhdr2_from_fl is invoked to attempt to fill the HDR2 portion of the file data from the file link. This step ensures that any file attributes specified in the attach description do, in fact, match the attributes recorded in the HDR2 label, if any. The procedure fill_flhdr2_from_fd is called to fill the HDR2 portion of the file link from the file data, <u>without</u> applying any defaults. This step ensures that a complete, consistent attribute set is composed from the attach description, the HDR2 label (if any), or both.

<u>Entry</u>: truncate_chains

This procedure is called from the $open code preparatory to creating, extending, modifying, or generating a file. Since the act of writing physically truncates the file set, the file and volume chains must be correspondingly truncated. This procedure is also invoked by consistent and abort_file to perform the same function should an output operation be abnormally terminated during opening or closing.

A cleanup handler is first established so that interrupting the truncation process does not leave the chains in an inconsistent state. The file chain is truncated immediately following the link pointed to by cseg.flP. This file link is referred to as the desired link. The value of cseg.flP is saved, so that it can be reset to point to this link once the truncation process is complete. The desired link at that point is the last link in the chain. If, therefore, the desired link is already the last link, no action need be taken and the procedure simply returns. Otherwise, cseg.flP is set to point to the next link in the chain (cseg.flP = cseg.flP -> fl.nextP). This link is referred to as the truncation link. The forward chain pointer in the desired link (cseg.flP -> fl.backP -> fl.nextP) is nulled, logically truncating the file chain. However, the truncation link as well as all following links (if any) are still physically allocated in cseg.chain_area.

If the truncation link is not an eofsl, its file link index (fl.flX) occurs in the range of its associated volume link (vl (fl.vlX).fflX $\leq$ fl.flX $\leq$ vl (fl.vlX).lflX). This volume link must therefore be either partially or completely truncated; i.e., reference to the truncation link and all subsequent file links (if any) must be removed. In addition, all subsequent volume links must be entirely truncated. Since an eofsl has no associated volume link, truncating the eofsl does not affect the volume chain at all. If the truncation link corresponds to the first file section on a volume (fl.flX = vl (fl.flX).fflX), that volume link is entirely truncated (vl.fflX, vl.cflX, vl.lflX = 0). Otherwise, the volume link must be partially truncated. The last file link index for the volume (vl (fl.vlX).lflX) is set to the desired file link index (vl (fl.vlX).lflX = fl.flX - 1), since that is the last file link on the volume. In either case, all subsequent volume links, if any, are entirely truncated.

Because the critical portion of the procedure is complete, the cleanup handler is reverted. Beginning with the truncation link, it and all subsequent links (if any) are freed from cseg.chain_area. cseg.flP is restored to point to the desired link.

<u>Entry</u>: vl_init

This procedure is called by next_volume to initialize a new volume link. Every member of the volume link structure except the volume name (vl.volname), is set.

<u>Entry</u>: vname

This function is called by another_volume to validate and normalize a user-supplied volume name. If the volume name is longer than 6 characters, it is invalid and the procedure returns "0"b. If it is exactly 6 characters, the procedure returns "1"b. If it is less than 6 characters, the name is normalized and the procedure returns "1"b. If the name is entirely numeric, it is normalized by right justifying and padding on the left with zeros to length six. If not entirely numeric, it is normalized by left justifying and padding on the right with blanks to length six.

<u>Entry</u>: write_HDRs

This procedure is called by the $open code to write HDR1 and HDR2 labels as part of the file creation and generation processes. It is also called by write_new_section to write the header labels for a new file section. It sets the file link trailer label type (fl.eox) to 0, indicating that the file section contains (as of yet) no trailer labels, and transfers to the common body of code.

Entry: write_EOFs

This entry point is called by the $close code to write an EOF trailer label set once file processing is complete. It is also called by abort_file to overwrite an EOV trailer set with an EOF trailer set, thus truncating one or more defective file section(s). It sets fl.eox to 1, indicating an EOF trailer set and that the file section is the last (or only) of the file. It then transfers to the common body of code.

Entry: write_EOVs

This entry point is called by the mainline data_eot code to write an EOV trailer label set once physical end of tape has been detected by the logical record I/O procedure. It sets fl.eox to 2, indicating an EOV trailer set and that the file section is medial.

The common code performs the actual label writing by invoking write_label. The procedure write_HDRs always writes a HDR2 label, but that write_EOFs and write_EOVs will only write an EOF2 or EOV2 if the file's header label set includes a HDR2. This practice ensures that the header and trailer label sets of files not created or generated by the I/O Module remain symmetric. The physical end of tape is ignored, so that volume switching is only driven by EOT during logical record I/O operations.

Entry: write_TM

This procedure is called by numerous internal procedures to write either 1 or 2 tape marks, adjusting the volume link intrafile position indicator (vl.pos) accordingly. The entry point tape_ansi_tape_io_$order is invoked to perform the actual tape mark write operation(s). For each tape mark written, vl.pos is incremented. This is done according to the rules of modulo 3 arithmetic, and an overflow causes the current file link index (vl.cflX) to be incremented. (There are only 3 possible intrafile positions.)

Entry: write_new_section

This procedure is called by the mainline data_eot code to add a new file section to a file. The variable cseg.flP is set to point to the next file link, which must be an eofsl, and make_eofsl_real is invoked to establish a file/volume link interrelationship. The procedure build_eofsl is called to append another eofsl to the file chain, and move_tape_ is called to position the volume set preparatory to writing the new section's header labels.

Before the labels can be written, however, the first HDR1 label on the volume must be checked to ensure that the volume's current contents are expired. If this is not the case, initialize_permitA is invoked to query the user for permission to overwrite. If permission is denied, the procedure returns the error code error_table_$unexpired_volume. Otherwise, initialize_volume is invoked to reinitialize and move_tape_ is called to reposition for writing.

The procedure fill_new_section_fl is called to fill the file link with data derived from the previous section's link, and write_HDRs is called to write the new header label set. The procedure write_TM is invoked to write one tape mark.

MODULE: tape_ansi_nl_file_cntl_

This module performs the iox_$open and iox_$close functions for IBM NL (nonlabeled) file sets. In addition, it performs end-of-file and end-of-volume processing for tape_ansi_ibm_lrec_io_, as well as beginning-of-file and end-of-file positioning for tape_ansi_position_.

Entry: tape_ansi_nl_file_cntl_$open

This entry point performs the iox_$open function. It positions to the attached file, mounts and/or demounts volumes as needed, and maintains the volume chain.

<u>Usage</u>

```
     dcl  tape_ansi_nl_file_cntl_$open ext entry
          (ptr, fixed bin, bit (1) aligned, fixed bin (35));

     call  tape_ansi_nl_file_cntl_$open (iocbP, mode, mbz, code);
```

See the description of tape_ansi_file_cntl_ for a discussion of the arguments.

If code is nonzero, an error has occurred and the I/O switch is not open. The following is a nonexhaustive list of the error_table_ codes that can be returned. See the description of tape_ansi_file_cntl_ for a discussion of their meanings.

```
          file_aborted
          file_busy
          incompatible_attach
          incompatible_encoding_mode
          insufficient_open
          invalid_block_length
          invalid_cseg
          invalid_file_set_format
          invalid_record_length
          noalloc
          no_file
          no_next_volume
          uninitialized_volume
```

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. If either cseg.invalid or cseg.file_lock is "1"b, an error exit is taken with the error code error_table_$invalid_cseg or error_table_$file_busy, respectively. A cleanup handler is established to ensure that neither the cseg nor the file is left in an inconsistent state. The opening mode is validated against the attach description. If a discrepancy exists, the procedure returns the error code error_table_$incompatible_attach.

The file link pointer (cseg.flP) is set to null, and the file data volume index (fd.vlX) is initialized to 1, for the first (or only) volume. The file chain is not used by this procedure, since it does not maintain a "history" of the file set. All data used to process a particular file are maintained in the file data structure. This being the case, it is necessary to index into the volume chain using a file data variable (fd.vlX), as opposed to the corresponding file link variable (fl.vlX), which is undefined.

If the opening is for sequential_input, the record format and block length must be specified. For all record formats except U, the record length must also be specified. The absence of any of the above attributes causes the procedure to return the error code error_table_$insufficient_open. If the character encoding mode is not specified, its default is EBCDIC. The procedure move is invoked to position to the file. If an error occurs, the procedure transfers to the label er_exit with whatever error code was returned. The code at er_exit calls consistent before returning to ensure that no inconsistencies exist in the file or the cseg. If the positioning is successful, lrec_open is called to perform final consistency checks on the file attributes and to initialize the logical record I/O control structure (cseg.lrec). If no error has been detected, the I/O switch is opened and the procedure returns.

If the opening is for sequential_output, the output mode must be create. (Nonlabeled files cannot be attached for extension, modification, or generation.) If the record format is not specified, its default is VB. If the block length is not specified, its default is 8192. An unspecified record length defaults to the block length if the record format is F or FB, to 8188 if the record format is V or VB, or to 1044580 (sys_info_$max_seg_size * 4) if the record format is VS or VBS. The default encoding mode is EBCDIC. The procedure move is called to position to the desired file. If an error occurs, the procedure transfers to er_exit with whatever error code was returned. The procedure lrec_open is called to perform the final attribute consistency checks and to initialize the logical record I/O structure. If this step succeeds, the I/O switch is opened and the procedure returns.

Entry: tape_ansi_nl_file_cntl_$data_eof

This entry point is called by tape_ansi_ibm_lrec_io_ when a tape mark is detected in the course of a read operation. It determines whether the EOF indicates the end of the file or merely the end of a file section. In the latter case, volume switching is performed.

Usage

```
dcl tape_ansi_nl_file_cntl_$data_eof ext entry
    (ptr, fixed bin (35));

call tape_ansi_nl_file_cntl_$data_eof (iocbP, code);
```

If code is neither zero nor error_table_$no_next_volume, an error has occurred and the I/O switch is closed. The latter code indicates that no additional volumes are available, and that further I/O activity should be prohibited.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. It is not necessary to check either cseg.invalid or cseg.file_lock. The former must be "0"b, or tape_ansi_ibm_lrec_io_ could not have been invoked, and the latter must be "1"b because it was invoked. The consistency code (cc) is set to zero and a cleanup handler is established.

Since the caller has already read over a tape mark (into the next file), the volume link's current position indicator (vl.cflX) must be incremented. (The variable vl.cflX does not strictly represent an index into the file chain, but rather the actual physical file number. The variable vl.pos is not used because there are no intrafile positions within a nonlabeled file, and vl.fflX and vl.lflX are not used because there is no file chain upon which to base a range of indices.) The entry point tape_ansi_ibm_lrec_io_$close is called to terminate I/O on the file section, synchronize the tape position, etc. If an error occurs during this process, consistent is invoked and the I/O switch is closed.

To read a multivolume file, the user must specify every volume set member in the attach description. Hence, determining whether the file section is the last (or only) one of the file set (and therefore whether or not volume switching is required) is a minor task. If the current volume index (fd.vlX) is equal to the index of the last volume (cseg.vcN), the file section is terminal and the procedure returns the status code error_table_$end_of_info. Otherwise, move is called to position to the next file section, which is by definition the first file on the next volume. If an error occurs while positioning, consistent is invoked and the I/O switch is closed. If positioning is successful, the procedure returns to tape_ansi_ibm_lrec_io_ to resume reading data.

Entry: tape_ansi_nl_file_cntl_$data_eot

This entry point is called by tape_ansi_ibm_lrec_io_ when end of tape is detected in the course of a data write operation, and by tape_ansi_control_ in response to an "feov" order. It switches to the next volume of the volume set, if any.

Usage

```
     dcl  tape_ansi_nl_file_cntl_$data_eot ext entry
          (ptr, fixed bin (35));

     call tape_ansi_nl_file_cntl_$data_eot (iocbP, code);
```

If code is neither zero nor error_table_$no_next_volume, an error has occurred and the I/O switch is closed. The latter code indicates that no additional volumes are available and that further I/O activity should be prohibited.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. For the reasons stated above, neither cseg.invalid nor cseg.file_lock need be checked. The procedure next_volume is invoked to determine whether or not another volume is available for continued processing. If not, the procedure returns the status code error_table_$no_next_volume.

If a volume is available, cc is set to 2 and a cleanup handler is established. This handler prevents leaving an inconsistent file section on the volume set should the procedure be prematurely terminated. The procedure write_TM is invoked to write a single tape mark, which logically terminates the file section. If an error occurs, consistent is invoked and the I/O switch is closed. Writing the tape mark leaves the file set in a consistent state. Therefore, the consistency code can be reset to 0, so that the file section is not truncated should an error occur in a subsequent step. The procedure move is called to position to the beginning of the next volume, where the new file section is recorded. If an error occurs, consistent is invoked and the I/O switch is closed. If positioning is successful, the procedure returns to tape_ansi_ibm_lrec_io_ to resume writing data.

Entry: tape_ansi_nl_file_cntl_$beginning_of_file

This entry point is called by tape_ansi_position_ to implement the -1 (position to beginning of file) operation.

Usage

```
     dcl  tape_ansi_nl_file_cntl_$beginning_of_file
          ext entry (ptr, fixed bin (35));

     call tape_ansi_nl_file_cntl_$beginning_of_file
          (iocbP, code);
```

If code is nonzero, an error has occurred and the I/O switch is closed.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. Neither cseg.invalid nor cseg.file_lock need be checked. The variable cc is set to 0 and a cleanup handler is established that calls consistent and closes the I/O switch. The procedure move is invoked to position to the first (or only) file section. If an error occurs, consistent is called and the I/O switch is closed.

Entry: tape_ansi_nl_file_cntl_$end_of_file

This entry point is called by tape_ansi_position_ to implement the +1 (position to end-of-file) operation.

Usage

```
     dcl  tape_ansi_nl_file_cntl_$end_of_file
          ext entry (ptr, fixed bin (35));

     call tape_ansi_nl_file_cntl_$end_of_file
          (iocbP, code);
```

If code is nonzero, an error has occurred and the I/O switch is closed.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. Neither cseg.invalid nor cseg.file_lock need be checked. The variable cc is set to 0 and a cleanup handler is established that calls consistent and closes the I/O switch.

If the file is already positioned past the end-of-file tape mark (as the result of tape_ansi_ibm_lrec_io_ read operations), tape_ansi_tape_io_$order is invoked to position immediately after the last data block (immediately preceding the tape mark). The current position indicator (vl.cflX) is decremented to reflect the new position, and the procedure returns.

Otherwise, tape_ansi_ibm_lrec_io_$close is called to synchronize the tape position before performing any other positioning operation. If the current volume is not the last of the volume set (fd.vlX ^= cseg.vcN), the current file section is not the last of the file. The procedure move is therefore invoked to position to the last section, which by definition must be the first file on the last volume. The entry point tape_ansi_tape_io_$order is invoked twice: first, to position immediately after the end-of-file tape mark, and second, to backspace immediately before it.

If any of the above procedure calls results in an error, consistent is invoked and the I/O switch is closed.

Entry: tape_ansi_nl_file_cntl_$close

This entry point performs the iox_$close function. In the read case, it merely terminates logical record I/O in a consistent manner and closes the I/O switch. In the write case, it writes the end-of-file and end-of-file-set tape marks and then closes the I/O switch.

Usage

```
     dcl  tape_ansi_nl_file_cntl_$close ext entry
          (ptr, fixed bin (35));
     call tape_ansi_nl_file_cntl_$close (iocbP, code);
```

If code is nonzero, an error has occurred. In the write case, the file (or a portion thereof) may be truncated. In either case, the I/O switch is always closed.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB. If the file is in use (cseg.file_lock = "1"b), the procedure returns the error code error_table_$file_busy. If cseg.invalid is "1"b, the I/O switch must be closed, but no operations can be performed on the file itself. The return code is set to error_table_$invalid_cseg and a cleanup handler is established that closes the I/O switch. The I/O switch is then closed.

If the I/O switch is open for sequential_input, cc is set to 0 and a cleanup handler is established that calls consistent and closes the I/O switch. The entry point tape_ansi_ibm_lrec_io_$close is invoked to synchronize the tape, the volume is rewound if the close_rewind order has been issued, and the I/O switch is closed.

If the opening was for sequential_output, cc is set to 2 and a cleanup handler is established that calls consistent and closes the I/O switch. The entry point tape_ansi_ibm_lrec_io_$close is invoked to synchronize the tape. If end of tape is detected, it is ignored, because the I/O module processes EOT only when writing data via the tape_ansi_ibm_lrec_io_$write_record entry. The procedure write_TM is called to write two tape marks (one for end-of-file, the other for end-of-file-set). EOT is similarly ignored. The variable cc is then reset to 0, since the file section is now valid, and the I/O switch is closed.

If an error occurs during any of the above procedure calls, consistent is invoked and the I/O switch is closed.

Internal Procedures

Entry: abort_file

This procedure is called by consistent when an unrecoverable error occurs during the processing of an output file. After the volume position (vl.cflX) is invalidated, write_TM is invoked to write two tape marks. If the tape marks are successfully written, the file set format is valid. An informatory message is written on user_output via ioa_ and the procedure returns. If the tape marks cannot be written, the file set format is invalid, and a message is issued to that effect. End-of-tape while writing the tape marks is ignored because EOT is processed only at write data time.

Entry: consistent

This procedure is called from multiple points within the module when an error occurs during file (as opposed to data) processing. It ensures that the control segment is always a valid model of the file set, and that the file set is self-consistent.

Depending upon the consistency code (cc), one of two actions is taken. For consistency codes 0 and 1, the current volume's current file position indicator (vl (fd.vlX).cflX) is invalidated and the procedure returns. (tape_ansi_nl_file_cntl_ retains the three-valued consistency code used by tape_ansi_file_cntl_, even though codes 0 and 1 result in identical actions.) For code 2, abort_file is called to ensure that the file set is left in a consistent state.

Entry: handler

This procedure is called by the on unit of the any_other condition handler established prior to IOCB manipulation. Because IPS interrupts are masked immediately after the on unit is established, this procedure should almost never be invoked. If it is invoked, one of two cases has occurred.

If the IPS mask is nonzero, interrupts have already been masked and none should have occurred. This is regarded as a fatal error, and the external procedure terminate_process_ is called to terminate the process. This drastic step is necessary to ensure that critical IOCBs are always valid. If the IPS mask is zero, the interrupt has occurred during the fraction of time between on unit establishment and IPS masking, and the interrupt is valid. The external procedure continue_to_signal_ is therefore called to pass the condition down the stack.

Entry: initialize_permitA

This function is called by move when a newly mounted volume is found to have a VOL1 label and the volume is targeted to receive output data (thus destroying the label). The external procedure command_query_ is invoked to query the user for permission to use the volume. If permission is granted, the procedure returns "1"b. If permission is denied, the procedure returns "0"b.

Entry: initialize_permitB

This function entry point in the procedure initialize_permitA is called by move when the first block of a newly mounted volume targeted to receive output data is found to be unreadable. Since it cannot be determined whether or not the volume is indeed labeled, the user must be queried for permission. The external procedure command_query_ is called to perform the query. If permission is granted, the procedure returns "1"b. If permission is denied, the procedure returns "0"b.

Entry: lrec_open

This procedure is called from the mainline open code as the last step in the opening process prior to IOCB manipulation. It performs final validity checks on the file attributes and initializes the logical record I/O control structure (cseg.lrec).

The block length is checked to ensure that it is not greater than 8192 (the present implementation restriction). If the opening mode is for sequential_output, the block length must also be greater than 18, and evenly divisible by 4. (The block length, effectively, must be at least twenty. The two separate constraints are enforced to distinguish between a standard requirement that blocks of 18 or fewer bytes are not permitted and an implementation restriction that only words can be written.) The remaining checks apply to block/record length interrelationships on a per-format basis. If any of the above steps fails, the procedure performs a nonlocal transfer to er_exit.

Entry: move

This procedure is called from multiple points within the module to perform the actual tape positioning function.

If the desired volume is not mounted and the user-specified device limit would not be exceeded, tape_ansi_mount_cntl_$mount is called to mount the volume on a newly assigned device. If the new device assignment cannot be completed because it would exceed the process's device limit, control is transferred to the remount algorithm described below.

If the user-specified device limit would be exceeded, the desired volume can only be mounted in place of some other volume, an operation termed remounting. The volume chain is searched to select the volume to be demounted, first from the first volume set member up to the desired volume, and then from the last volume set member down to the desired volume. The search algorithm is optimized for the most usual case of sequential volume processing, and a candidate volume is always found. The entry point tape_ansi_mount_cntl_$remount is called to demount the candidate volume and to (re)mount the desired volume on the same device.

If the desired volume is already mounted when move is invoked, none of the above steps need be done. In any case, the drive number (currently stored in vl.rcp_id) and event channel id (vl.event_chan) of the (now) current volume are stored in the tseg portion of the cseg. When rcp_ and tape_ioi_ replace tdcm_ as the device interfaces, these two steps will have to be modified. It is likely, however, that the rcp_id, the event channel, and the tape_ioi_ id of the current volume will still be maintained in the cseg outside the volume chain.

The volume index maintained in the file link (fl.vlX) is set to the volume index of the current volume. It is this step that actually makes a volume the "current volume", since all references for positioning and volume activity (outside of move) are volume link references of the form vl (fl.vlX).

If the opening mode is sequential_output, a number of checks must be performed before the volume can be used. If the VOL1 label status code (vl.write_VOL1) indicates either a blank tape or no VOL1 label, the volume can be used without further checking. Any other code indicates the presence (actual or possible) of a VOL1 label that cannot be overwritten unless certain criteria are satisfied.

If the file to be written is not the first file, the volume cannot possibly be reformatted as a nonlabeled volume. The external procedure ioa_ is called to write an informatory message on user_output, and the procedure returns the error code error_table_$uninitialized_volume. If it is the first file, initialization can be possible. Based upon the value of vl.write_VOL1, either initialize_permitA or initialize_permitB is called to query the user for permission. If permission is granted, tape_ansi_tape_io_$order is called to rewind the tape and write_TM is called to overwrite the VOL1 label with two tape marks. The variable vl.cflX is set to indicate the new current file position (3), and vl.write_VOL1 is set to indicate the absence of a VOL1 label.

Volume positioning complete, the procedure positions to the desired file. If the current position is unknown (vl.cflX = 0), tape_ansi_tape_io_$order is called to rewind the volume and vl.cflX is set to 1. If the volume is positioned before the desired file (vl.cflX < fX, the desired file index), the difference is computed and tape_ansi_tape_io_$order is invoked to forward space the appropriate number of tape marks. If blank tape is detected, the desired file does not exist. The error code error_table_$no_file is returned.

If the volume is positioned after the desired file (vl.cflX > fX), a combination of tape_ansi_tape_io_$order calls to backspace and forward space files is issued to effect the desired positioning. If the volume is positioned at the desired file (vl.cflX = fX), tape_ansi_tape_io_$order calls are issued to ensure that the volume is positioned to the first block of the file and not to an intermediate, indeterminate position.

If none of the above steps results in an error, vl.cflX is set to fX and the procedure returns. If an error occurs during any step, vl.cflX is set to 0 and the procedure returns whatever error code was set.

Entry: next_volume

This function is called by the data_eot entry to determine whether or not another volume is available for concatenation to the volume set. It is called only in the output csincee, since the volume set membership for the input case is determined entirely by the volume list specified in the attach description.

If the current volume is not the last of the volume chain (fd.vlX < cseg.vcN), the next volume exists and the procedure returns "1"b. If the current volume index is 63, the implementation maximum, the external procedure ioa_ is invoked to issue an informatory message on user_output and the procedure returns "0"b. If neither of the above cases is satisfied, another_volume is called to query the user for the next volume name, if any. If none is supplied, the procedure returns "0"b. If one is supplied, the volume chain is extended (cseg.vcN is incremented), vl_init called to initialize the new link, and the link's volume name is set. The procedure returns "1"b.

Entry: vl_init

This procedure is called by next_volume to initialize a new volume link. Every structure member except the volume name is set appropriately to a logically null value.

Entry: vname

This function is called by another_volume to validate and normalize a volume name. A volume name must be six characters or fewer. If it is longer than six characters, the procedure returns "0"b. Otherwise, there are two normalization cases. If the name is entirely numeric, it is padded on the left with zeros to length six. If it is not entirely numeric, it is padded with blanks on the right to length six. The procedure returns "1"b.

Entry: another_volume

This function is called by next_volume to determine whether or not a user-specified volume is to be concatenated to the volume set. It is only invoked if end of tape (EOT) is detected during a data write operation.

The external procedure command_query_ is invoked to query the user as to whether or not processing is to be continued on another volume. If processing is not to continue, the procedure returns "0"b. If processing is to continue, command_query_ is called again to obtain the name of the volume, along with an optional mount message. The supplied volume name is validated by calling vname. If it is invalid, command_query_ is invoked again to obtain a valid name. If the comment is invalid, command_query_ is similarly invoked. Once a valid volume name (and optional comment) is obtained, the procedure returns "1"b.

Entry: write_TM

This procedure is called to write either one or two tape marks, thus terminating a file section or file set, respectively. The entry point tape_ansi_tape_io_$order is invoked to write the tape mark(s) and the volume's current file position (vl.cflX) is incremented for each tape mark written. If EOT is detected it is ignored, so that all volume switching takes place at data write time.

MODULE: tape_ansi_detach_

This module performs the iox_$detach function for both tape_ansi_ and tape_ibm_. It performs resource disposition as specified in the attach description, issues a volume set status message if necessary, and manipulates the IOCB to indicate the detached state.

Usage

```
     dcl  tape_ansi_detach_ ext entry (ptr, fixed bin (35));

     call  tape_ansi_detach_ (iocbP, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	code	is a standard status code. (Output)

If code is error_table_$file_busy, the I/O switch is not detached. If code is zero or any other nonzero value, the I/O switch is detached.

Internal Logic

The cseg pointer is obtained from the IOCB and cseg.file_lock is tested to ensure that an I/O operation is not in progress. If the file is locked, the procedure returns the error code error_table_$file_busy. Otherwise, the file is locked and a cleanup handler is established. If the cseg pointer is null when the cleanup handler is invoked, the IOCB is detached. No resource disposition or volume set status activity is possible. If the cseg pointer is nonnull, the detach operation is performed as though the specified disposition were "-retain none". Thus, a quit while detaching always results in a detached IOCB, though resource disposition may or may not be performed as specified in the attach description.

One of five possible resource disposition functions is then performed. Currently, only two of the functions are distinct: retention of no resources, and retention of all resources. When the I/O module is converted to call rcp_ directly, the default rcp_ retention function can be added. When rcp_ is further enhanced to provide individual resource management of both devices and volumes, the additional disposition options of device retention and volume retention can be implemented.

If no resources are to be retained, cseg.write_ring and cseg.protect are set to "0"b. (Since all devices are unassigned, they can neither have volumes mounted with rings nor be file protected.) The volume chain is then scanned for links having nonzero vl.rcp_id values. Such a value indicates an assigned device. Such a link's vl.cflX is invalidated, and tape_ansi_mount_cntl_$free is called to unassign the device. If an error occurs during unassignment, cseg.invalid is set to "1"b and the scan continues. When all links up to and including vl (cseg.vcN, the last active link) have been checked, control transfers to perform the I/O module's internal detach-state functions. If all resources are to be retained, cseg.invalid is checked to ensure that the I/O module is capable of performing another attachment. If not, control is transferred to the resource unassignment code.

There are three steps to be performed when the I/O module enters the detached state. First, if an iox_$read_length operation was ever performed, the temporary buffer segment in the process directory must be truncated. Therefore, if cseg.rlP is nonnull, the external procedure hcs_$truncate_seg is invoked to truncate the segment and cseg.rlN (the segment's character count) is set to -1, indicating no record in the buffer.

Second, it must be determined whether or not the control segment is internally consistent, hence usable in a subsequent attachment. If cseg.invalid is "0"b, the cseg is usable. Otherwise, the cseg must be deleted so that a subsequent attachment (if any) makes an entirely new cseg, file chain, etc. The external procedure hcs_$delentry_seg is invoked to delete the control segment and the read length buffer segment (if any).

The third and last step is performed only if the cseg is valid, volumes have been demounted, and write rings were in place. Under this combination of circumstances, it is possible that the volume set membership of a multivolume file or file set changed during the course of the attachment. Since the user may not know exactly how many volumes are included in the volume set, an informative message is issued.

If the file set is IBM nonlabeled, a test is made to determine whether the last volume processed (fd.vlX) is the last volume of the volume chain (cseg.vcN). If so, no message need be issued because the entire volume set membership must have been specified either in the attach description, or by the user via the command_query_ facility. If not, an informative message is issued and the volume chain is truncated to the last volume set member (cseg.vcN = fd.vlX). This is done to ensure that volumes that are not volume set members are not considered as such in subsequent attachments.

If the file set is either ANSI or IBM SL, it is determined whether or not the last volume in the volume chain contains a file section. If it does, the volume set membership comprises all volumes and no message need be issued. If it does not, the volume chain is scanned to find the last volume set member, and the volume chain is truncated at that point. An informative message is then issued.

Finally, the IOCB is manipulated to indicate the detached state. If the cseg still exists (it normally does, unless it was deleted previously due to an inconsistency), the file and file set locks (cseg.file_lock and cseg.file_set_lock) are set to "0"b and the read length buffer segment (if any) is terminated by calling the external procedure hcs_$terminate_noname.

MODULE: tape_ansi_lrec_io_

This module performs the iox_$read_record and iox_$write_record functions for ANSI file sets.

Entry: tape_ansi_lrec_io_$read_record

This entry point performs the iox_$read_record function.

Usage

```
     dcl  tape_ansi_lrec_io_$read_record ext entry (ptr, ptr,
          fixed bin (21), fixed bin (21), fixed bin (35));

     call tape_ansi_lrec_io_$read_record (iocbP, ubP,
          buf_len, rec_len, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	ubP	is a pointer to the user's record buffer. (Input)
3.	buf_len	is the number of characters to be read. (Input)
4.	rec_len	is the number of characters actually read. (Output)
5.	code	is a standard status code. (Output)

The following is a nonexhaustive list of error_table_ codes that can be returned:

file_busy	file in use for other I/O activity; no data returned.
fatal_error	unrecoverable error occurred; all, some, or no data returned. Data can be incorrect.
long_record	actual record length exceeded buf_len (requested length); buf_len characters returned, remainder of record discarded.
invalid_record_desc	a variable-length or spanned record's RCW or SCW is invalid; some or no data returned. Data can be incorrect.
tape_error	a parity error occurred while reading; all, some, or no data returned. Data can be incorrect.

It is important to note that for the blocked record formats, error_table_$tape_error is returned with the first record of the block that contains the error. Since a parity error is associated with a physical block as opposed to a logical record, the first record may or may not contain the invalid character or characters. If subsequent iox_$read_record calls are made, records from the same block can contain the invalid data even though their return codes are zero.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB and cseg.file_lock is checked to ensure that the file is not in use. If the file is in use, the status code error_table_$file_busy is returned. If the file is not in use, a cleanup handler is established and the file is locked. If invoked, the cleanup handler unlocks the file lock (cseg.file_lock) and sets the logical record I/O lock (cseg.lrec.code) to error_table_$fatal_error. This step is necessary because an interrupted logical I/O operation can leave the internal I/O buffers and logical record processing variables in an inconsistent state. The logical record I/O lock is checked and If it is nonzero, the procedure immediately returns that error code.

The desired record may have already been read as the result of an iox_$read_length call. If so, the read_length buffer count contains a valid value (cseg.rlN ^= -1). If the user's request (buf_len) is equal to or greater than the number of characters in the buffer (cseg.rlN), cseg.rlN characters are returned with status code zero. If buf_len is less than cseg.rlN, buf_len characters are returned with status code error_table_$long_record. The appropriate number of characters are moved into the user's buffer from the read_length buffer and rec_len is set to the number of characters moved. The variable cseg.rlN is set to -1 to indicate that the read_length buffer no longer contains a valid record. The logical record count (cseg.lrec.reccnt) is incremented, the file is unlocked, and the procedure returns.

If the record is not in the read_length buffer, control is transferred to one of the four format routines. Three automatic variables are used by all four routines to control their operation. The variable remain is set by the internal procedure get_record and contains the number of characters in the block that remain to be processed. The variable move is set by the format routines to the number of characters moved to the user's buffer by the internal procedure move_to_user. The variable req_off is set to the number of characters processed by a single logical record request. Its value can differ from that of move and is used by the internal procedure read_release both to locate the beginning of the next record within a block and to release the I/O buffer when the block is exhausted.

For U format, get_record is called to obtain a record. The variable move is set equal to remain, because a U format record fills an entire block including pad characters (if any). If buf_len is less than move, the long record switch (the automatic variable long_record) is set to "1"b and move is set to buf_len so that the number of characters returned is equal to the number requested. (In the absence of any other I/O error or event, long_record = "1"b at exit time causes the procedure to return the status code error_table_$long_record.) Since each logical record request requires a new block, req_off is set equal to remain so that the I/O buffer is released. The procedure move_to_user is called to move the record to the user's buffer and read_release is called to release the I/O buffer. Control then passes to the normal exit routine.

For F and FB format, get_record is called to obtain a record. If the file's record length (fd.reclen) exceeds remain, a short record situation exists. Since iox_ does not treat this case as an error, move is set equal to remain without setting a status code. If fd.reclen is less than or equal to remain, move is set equal to fd.reclen so that only one record's worth of data is moved. If buf_len is less than move, the user's buffer is too small to contain all the available data. The long record switch is therefore set to "1"b and move is set equal to buf_len so that only the requested number of characters is moved. The variable req_off is set to fd.reclen because each logical record request must process an entire record, even if only a portion of that record is actually moved to the user's buffer. The procedure move_to_user is called to move the data and read_release is called to position beyond the record. (In F format, the I/O buffer is released after each record is processed; In FB format, it is only released after the last record in a block has been processed.) Control then passes to the normal exit routine.

For D and DB format, get_record is called to obtain a record. If a block pad character (circumflex, "^") is found where the RCW should be, the remainder of the block contains no valid data. The entry point tape_ansi_tape_io_$release_buffer is called to release the I/O buffer and control passes back to the get_record call. Once a record has been obtained, a pointer to the record's RCW (record control word) is made and the record's actual length is extracted into the automatic variable data_len. If the data length cannot be extracted, or if it exceeds the number of characters remaining in the block (remain - 4), control passes to the invalid record descriptor error exit. If the RCW is valid, move is set equal to data_len. If, however, buf_len is less than move, long_record is set to "1"b and move is reset equal to buf_len. The variable cseg.lrec.offset, the current processing offset within the I/O buffer, is incremented by 4 (the length of an RCW) so that the RCW is not processed as part of the data. The variable req_off is set equal to data_len so that the request processes the entire record, even if only a portion is actually being returned. The procedure move_to_user is called to move the data to the user's buffer. The procedure read_release is called to position beyond the record. Control then passes to the normal exit routine.

For S and SB format, get_record is called to obtain a record segment. To keep track of the number of characters that have yet to be moved into the user's buffer to satisfy the request, the automatic variable left is initialized to buf_len. process_sw is then invoked to process and validate the segment's SCW (segment control word) and to extract the segment's data length into data_len. If left is greater than or equal to data_len, all the data in the segment is needed and move is set equal to data_len. If left is less than data_len, only a portion of the segment is needed to satisfy the (balance of the) request. In this case, long_record is set equal to "1"b and move is set equal to left. The procedure move_to_user is invoked to move the data to the user's buffer, left is decremented by move to equal the number of characters still required to complete the request, and read_release is called to position beyond the record segment.

The SCW type code is then checked. If the code indicates either a complete or terminal record segment, the entire logical record has been processed and control passes to the normal exit routine. If not, the remaining record segments must either be skipped (if the user's request is satisfied), or processed (if their data is needed to complete the request). In the latter case, left is nonzero. The procedure get_record is called to obtain the next record segment and control is passed back to the process_sw call described above. In the former case, left is zero. The procedure skip_segments is called to position beyond the last segment of the record. Control then passes to the normal exit routine with long_record set to "1"b because the user requested fewer characters than the record contains.

The normal exit routine increments the logical record count (cseg.lrec.reccnt). If a parity error has occurred, the return code is set to error_table_$tape_error. Otherwise, it is set to zero or whatever error code has been set by a previous step. If the return code is zero and long_record is "1"b, the return code is set to error_table_$longrecord. An error code therefore overrides the reporting of the long record condition. The variable rec_len is set equal to the automatic variable total, whose value has been maintained by move_to_user to be the total number of characters placed in the user's buffer. The variable cseg.file_lock is set to "0"b and the procedure returns.

The error and invalid record descriptor exit routines perform the same functions as described above, with the exception of incrementing cseg.lrec.reccnt.

Entry: get_record

This procedure makes a logical record available to the record format routines, either by reading a new block into an I/O buffer, or by setting the buffer processing variables for the next record already in a buffer.

If the I/O buffer pointer (cseg.lrec.bufP) is nonnull, at least one record is already in the I/O buffer. The variable remain is set to the number of characters not yet processed and the procedure returns. If cseg.lrec.bufP is null, tape_ansi_tape_io_$read is called to read a block. The variable cseg.lrec.bufP is set to point to the I/O buffer and cseg.lrec.nc_buf is set to the number of characters read. If the call returns a nonzero code, there are two main possibilities.

If the code is error_table_$eof_record, an end-of-file mark has been read. The entry point tape_ansi_file_cntl_$data_eof is invoked to determine whether the actual end of the file has been reached or whether the file is continued on another volume. If a zero code is returned, the file is continued on the next volume. Since tape_ansi_file_cntl_ has performed all necessary volume switching functions, control is simply passed back to the tape_ansi_tape_io_$read call. If the code is nonzero, either no more data exists or an error has occurred, and control passes to the exit routine.

If tape_ansi_tape_io_$read returns any other nonzero code, a parity or fatal error has occurred. If the code is error_table_$tape_error, the parity error switch (the automatic variable "parity_error") is set to "1"b and the current iox_$read_record operation is continued. Any other code causes control to pass to the error exit routine. (The iox_$control operation "reset_error_lock" can be used to permit further iox_$read_record calls, if and only if the lock value is equal to error_table_$tape_error.)

If the code was zero or error_table_$tape_error, the block count (cseg.lrec.blkcnt) is incremented and cseg.offset is set to the file's buffer offset value (fd.bo). This causes the block prefix (if any) to be skipped. As ANSI blocks can be padded to any length with circumflex characters ("^"), it is necessary to eliminate them (logically) from the I/O buffer. If cseg.nc_buf exceeds the desired block length (fd.blklen), the excess can be eliminated easily by setting cseg.nc_buf to fd.blklen. This step also ensures that no more characters can be extracted from a block than have been specified. Since U format blocks are processed with pad characters (if any) and D, DB, S, and SB records contain explicit data lengths, no further processing is necessary. The variable remain is set to the number of characters available for processing (cseg.nc_buf - cseg.offset) and the procedure returns.

For F and FB format, a further pad stripping algorithm must be applied. The number of records in the block is computed by dividing the number of possible data characters in the block (cseg.nc_buf - fd.bo) by the number of characters in a record (fd.reclen). The number of characters (if any) that do not fill a complete record is computed by taking the number of possible data characters modulo the record length. If these characters are all pad characters, they are eliminated (logically) by decrementing cseg.nc_buf. If any are not pad characters, they are as a group considered to form a short record, remain is set, and the procedure returns. If characters not contained in a complete record are not found, or if such characters are all padding, it is possible that additional padding exists. Starting with the last record in the block, each record is tested to determine whether it is all pad characters. Each record of padding causes cseg.nc_buf to be decremented by fd.reclen. The first record that is not padding causes remain to be set and the procedure to return. Eventually, remain is set to the number of characters available for processing.

Entry: process_sw

This procedure is called by the S and SB format routine to validate and process an SCW. If the first character of what ought to be an SCW is found to be a pad character, tape_ansi_tape_io_$release_buffer is called to release the I/O buffer. The procedure get_record is then called to obtain a record segment from the next block and control passes back to the pad checking code described above. Once an SCW has been obtained, the segment's data length is extracted into data_len, its type code is validated, and data_len is checked against the actual number of characters remaining in the block. An inconsistency detected by these checks results in a nonlocal transfer to the invalid record descriptor error exit. If the SCW is valid, cseg.offset is incremented by 5 (the length of an SCW) so that the SCW is not processed as part of the segment's data. The variable req_off is set equal to data_len.

Entry: skip_segments

This procedure is called by the S and SB format routine to skip record segments that are not required to satisfy the user's request (buf_len < total record length). The procedure get_record is called to obtain a record segment, and process_sw is called to process and validate its SCW. If the type code is that of a final segment, read_release is called to position beyond it and the procedure returns. Otherwise, read_release is called and control passes back to the get_record call. This algorithm is continued until the final segment is encountered and skipped.

Entry: move_to_user

This procedure is called by all four format routines to move data from the I/O buffer to the user's buffer. If move is zero, no data is to be moved and the procedure returns. Otherwise, a pointer is made to the first character to be moved from the I/O buffer, and another pointer is made to the location within the user's buffer where that character is to be placed. If the encoding mode (fd.mode) is not EBCDIC, no character conversion need be performed and the data is simply moved. Otherwise, ebcdic_to_ascii_ is invoked to translate and move the data. The automatic variable total is incremented by the value of move, to maintain a count of the total number of characters moved.

Entry: read_release

This procedure is called to release a logical record or record segment from an I/O buffer. If the record is the last (or only) one in the buffer, the entire buffer is also released. The variable cseg.offset (the current processing offset within the buffer) is incremented by the value of req_off (the number of characters processed by the format routine). The variable remain is calculated by subtracting the new current offset from the buffer character count (cseg.nc_buf). (The variable remain can become negative.) If the record format is S or SB and remain is less than 5, the I/O buffer is released, because the remaining characters are too few to be even the SCW of a zero-length record segment. If remain is greater than or equal to 5, the procedure returns.

For all other formats, if remain is less than 4, usually the I/O buffer is to be released. For U format this is always the case, and similarly for D and DB format because four characters is insufficient for even the RCW of a zero-length record. In these cases, the buffer is released and the procedure returns. For F and FB format, however, the buffer is only released if the logical record length is greater than the value of remain. If the logical record length is less than or equal to remain, the procedure simply returns. This practice causes the loss of short records (short record length < record length < 4) in some unusual, but possible, cases. Unfortunately, it is the only way to avoid processing the pad bytes (octal value 000) appended to blocks that have lengths not evenly divisible by 4. This ambiguity is built into the current software interface (tdcm_) to the MTS500 hardware and should no longer be a problem when the proposed interface (tape_ioi_) is implemented. The I/O buffer is released by calling tape_ansi_tape_io_$release_buffer.

Entry: tape_ansi_lrec_io_$write_record

This entry point performs the iox_$write_record function for ANSI file sets.

Usage

```
     dcl  tape_ansi_lrec_io_$write_record entry (ptr,
          ptr, fixed bin (21), fixed bin (35));

     call tape_ansi_lrec_io_$write_record (iocbP,
          ubP, buf_len, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	ubP	is a pointer to the user's buffer. (Input)
3.	buf_len	is the number of characters to be written. (Input)
4.	code	is a standard status code. (Output)

The following is a nonexhaustive list of error_table_ codes that can be returned.

file_busy	file in use for other I/O activity; record not written.
fatal_error	unrecoverable error occurred; see Write Errors below.

long_record	buf_len exceeds the maximum record and/or block length; the record is not written.
eov_on_write	no more records can be written on the current volume. For S and SB format, the record may be partially written; for all other formats, the record is not written.
tape_error	a parity I/O error has occurred; see "Write Errors" below.

WRITE ERRORS

In the case of a fatal or parity error, more records can be affected than just the particular record being written when the error code is returned. It is important to note that such an error is detected upon the writing of a block, and that each iox_$write_record call does not necessarily cause a block to be written. Hence, a zero status code does not guarantee that a record has been written at all, let alone written correctly. FB format blocks always, and DB and SB blocks can, contain multiple records, so that an error in writing a block affects every record packed into the block.

In addition, tape_ansi_ does not wait to check the status of a write operation after the operation has been issued. In the time between issuing a write operation and receiving its status, a number of further write operations can be issued. This method of operation is termed asynchronous processing and normally is highly satisfactory. If an I/O error occurs, however, not only is the erroneous block not written, but all blocks queued for writing subsequent to the error block are not written. Even in this case it is possible to maintain an accurate block count, but since the number of records per block can vary, an accurate record count cannot be maintained. Since U, F, and D format place only one record per block, the actual number of records written equals the block count (available by calling iox_$control "file_status" operation). For FB, DB, and SB format, an indeterminate number of records packed into blocks subsequent to the error block are not written, and for S and SB format, the record can have been partially written in blocks prior to the error block.

The cseg pointer is obtained from the IOCB and the cseg.file_lock is checked to be sure that the file is not busy for other I/O activity. If it is busy, the procedure immediately returns the error code error_table_$file_busy. Otherwise, a cleanup handler is established and the file lock is locked. If invoked, the cleanup handler unlocks the file lock and sets the logical record I/O lock to error_table_$fatal_error. This step is necessary because an interrupted logical I/O operation can leave the I/O buffer and its processing variables in an inconsistent state.

The logical record I/O lock (cseg.lrec.code) is checked to ensure that I/O has not been inhibited due to an unrecoverable error. If the cseg.lrec.code is nonzero, the return code is set to the logical I/O lock value, the file lock (cseg.file_lock) is unlocked, and the procedure returns. If all is well, the intrafile position indicator (vl (fl.flX).pos) is checked to determine whether or not the tape is positioned in the data portion of the file. The first time iox_$write_record is called subsequent to an opening, the tape is positioned in the file's header label group. In this case, tape_ansi_file_cntl_$position_for_output is called to write the header label group tape mark, such action defining the transition into the data portion of the file. If an error occurs while writing this tape mark, cseg.lrec.code is set to the error code value and control passes to the error exit routine. The tape mark is not written until the first logical record call for the following reason. The ANSI standard requires volume switching to be performed if end-of-tape is detected while writing the header label group. This causes a null file section to be recorded on the old volume. Yet if the I/O switch is then closed without an intervening I/O operation, another null file section is written on the new volume. By inhibiting the detection of end-of-tape until the header label group tape mark is written (this is not a Standard violation), and by delaying writing the tape mark until the first write operation, volume switching can be avoided if no write operations are issued. The close call causes just a single null file section to be written on the current volume.

Control then passes to one of the four record format routines. For U format, buf_len is checked to ensure that it does not exceed the maximum number of characters that can fit in a block (fd.blklen - fd.bo). If its value is too large, control passes to the long record error exit. The procedure get_buf is called to obtain an I/O buffer and move (the number of characters to be moved from the users buffer) is set equal to buf_len. The variable req_off (the number of characters to be written by this request) is also set equal to buf_len, and move_to_buf is called to move the user's data into the I/O buffer. The procedure write_buf is called to write the block, and control passes to the normal exit routine.

For F and FB format, buf_len is checked to ensure that it does not exceed the record length (fd.reclen). If it does, control passes to the long record error exit routine. The procedure get_buf is called to obtain an I/O buffer, if necessary. (For F format, an I/O buffer is obtained for each call, because each record requires a new block.) The variable move is set equal to buf_len and remain is set to the number of pad characters that must be appended to the user's data to make a complete record (fd.reclen - buf_len). This step is necessary because fixed-format records must be of identical lengths. If remain is nonzero, the appropriate number of blanks are inserted into the I/O buffer. The variable req_off is set equal to fd.reclen because each request processes a complete record, even if buf_len is less than fd.reclen. The procedure move_to_buf is called to move the user's data into the I/O buffer immediately before the inserted padding (if any). If records are not blocked (F format), write_buf is called to write the record. Otherwise, write_buf is not called unless the block contains as many records as can fit (cseg.offset = fs.blklen). Control then passes to the normal exit routine.

For D and DB format, data_len (the length of the record) is set to buf_len plus 4 (the length of an RCW). The value of data_len is checked to ensure that it does not exceed fd.reclen. If it does, control passes to the long record error exit. The procedure get_buf is called to obtain an I/O buffer, if necessary. (An I/O buffer is always obtained for D format, because each record requires a new block.) For DB format, it must be determined if the record to be written can fit into the current block, or if a new block is required. If data_len exceeds the number of remaining characters in the block (fd.blklen - cseg.offset), then write_buf is called to write the current block and get_buf is called to obtain a new I/O buffer. Either way, a pointer is made to the I/O buffer location where the record's RCW is to be constructed and the RCW is inserted. The variable cseg.offset is incremented by 4 so that the RCW is considered when computing the total block length, and req_off is set equal to buf_len. The variable move is also set equal to buf_len and move_to_buf is called to move the user's data. If records are not blocked, write_buf is called to write the record. Otherwise, write_buf is not called unless another record cannot fit in the current block (fd.blklen - cseg.offset < 4, where 4 is the length of a zero-length record). Control then passes to the normal exit routine.

For S and SB format, buf_len is checked to ensure that it does not exceed fd.reclen. If it does, control passes to the longer record error exit. The procedure get_buf is called to obtain an I/O buffer, if necessary. (An I/O buffer is always obtained for S format, because each record segment requires a new block.) The variable left contains the number of characters still to be moved from the user's buffer, and is initialized equal to buf_len. As each record segment is written, left is decremented by the number of characters written in that segment.

The variable remain is set to the number of characters remaining in the current block (fd.blklen - cseg.offset). A pointer is made to the I/O buffer location where an SCW is to be constructed.

The type code set in the SCW is dependent upon the amount of data still to be written (left) and the available space in the current block (remain). If left + 5 (the balance of the user's request plus 5 characters for the SCW) can fit in the block, the record segment is either a complete or final segment. If no data from the record has been previously placed into another segment, the type is complete; i.e., the segment contains the entire record. If some data has been placed into another segment, then the type is final; i.e., the segment is the last of a group of segments that _in toto_ make up the record. In either case, move is set equal to left, since the data to be moved into the segment is the balance of the request. If left + 5 characters cannot fit into the block, the record segment is either an initial or medial segment. If no data from the record has been previously placed into another segment, the type is initial; i.e., the segment is the first of a group of segments that _in toto_ make up the record. If some data has been placed into another segment, the type is medial; i.e., the segment is one of a group of three or more segments (but neither the first nor the last) that _in toto_ make up the record. In either case, move is set equal to remain - 5, so that as much data as will fit into the block is moved, leaving room for the 5 character SCW.

The variable left is decremented by the value of move, giving the amount of data (if any) to be moved into subsequent segments. The variable data_len, the actual record segment length, is set equal to move + 5 (to include the SCW) and is inserted into the SCW. The variable cseg.offset is incremented by 5, so that the SCW is considered when computing the total block length. The variable req_off is set equal to move and move_to_buf is called to move the user's data into the I/O buffer. The variable remain is set to the number of characters remaining in the block (remain - data_len). If record segments are not blocked (S format), control passes to write the I/O buffer. If record segments are blocked, the I/O buffer is only written if another nonzero length record segment could not fit into the block (remain < 6).

If the I/O buffer is to be written, write_buf is called to write it, get_buf is called to obtain another, and remain is set to the number of characters available in the new buffer (fd.blklen - cseg.offset). Whether or not the I/O buffer was written, left is checked to determine whether or not the user's request has been satisfied. If left is nonzero, control passes back to make another SCW pointer for the next record segment. If left is zero, control passes to the normal exit routine.

The normal exit routine first increments the logical record count (cseg.lrec.reccnt) and then sets the return code and logical I/O lock to the code returned by the last I/O operation (normally zero). If csw (the tape_ansi_lrec_io_$close entry switch) is "1"b, control passes to that entry's exit routine. Otherwise, the file lock is unlocked and the procedure returns. (The variable csw is initialized to "0"b upon procedure block activation, but is set to "1"b by the tape_ansi_lrec_io_$close entry.) The long record and error exits perform similar functions, with the exception of incrementing cseg.lrec.reccnt.

INTERNAL PROCEDURES

Entry: get_buf

This procedure is called to obtain an I/O buffer, if one is needed. If the I/O buffer pointer (cseg.lrec.bufP) is nonnull, a buffer is available and the procedure returns. If it is null, tape_ansi_tape_io_$get_buffer is called to make an I/O buffer available. The current offset equal within the buffer (cseg.offset equal) is set equal to the buffer offset length (fd.bo) to reserve space for a block prefix (if any). If the buffer offset length is nonzero, a block prefix of all blanks is inserted.

Entry: move_to_buf

This procedure is called to move data from the user's buffer to the I/O buffer. If move is zero, there is no data to be moved. In this case, cseg.offset is incremented by the value of req_off (the number of characters processed by the request) and the procedure returns. (The variable cseg.offset must be incremented to allow for the case of zero-length records in D, DB, S, and SB format. Such records consist of RCWs or SCWs alone.) If data is to be moved, pointers are made to the offset within the I/O buffer where the data is to be placed, and to the offset in the user's buffer from which the data is to be taken. If the encoding mode (fd.mode) is either ASCII or binary, the data is moved. If the mode is EBCDIC, ascii_to_ebcdic_ is called to translate and move the data. The variable total is incremented by the value of move, to maintain a count of the total number of characters moved. The variable cseg.offset is incremented by the value of req_off.

<u>Entry</u>: write_buf

This procedure writes a block, appending block pad characters if necessary. If cseg.offset is less than 20, the block must be padded. This is necessary for two reasons: 1) blocks of fewer than 18 characters must not be written, and 2) blocks to be written must consist of an integral number of words (4 characters/word). The number of pad characters is computed by subtracting cseg.offset (the number of characters presently in the block) from 20, and control passes to perform the padding.

If cseg.offset is greater than or equal to 20 but not evenly divisible by four, the block must still be padded to satisfy requirement 2) above. Padding for both cases is performed by inserting the appropriate number of pad characters into the I/O buffer immediately following its current contents, and then incrementing cseg.offset to reflect the new, adjusted block length.

The entry point tape_ansi_tape_io_$write is called to write the block. If the return code is zero, the block count (cseg.lrec.blkcnt) is incremented and the procedure returns. If the return code is nonzero, there are two major possibilities. If the code is not error_table_$eov_on_write, an error has occurred. The block count is decremented if more than one block was not written (cseg.blkcnt = cseg.blkcnt - cseg.soft_status.nbuf + 1). (The suspended buffer count is currently obtained directly from the cseg. Eventually, when tape_ioi_ becomes the device interface, a tape_ioi_ status entry is called to obtain this value.) The logical record count is invalidated by setting it negative, and control passes to the error exit.

If the code is error_table_$eov_on_write, end-of-tape has been detected. This is more in the nature of an event than an error. The variable cseg.lrec.blkcnt is incremented, because the block has been successfully written. If csw is "1"b (i.e., the procedure was entered at the $close entry point), the procedure simply returns. This is done so that EOT detection at close time does not force volume switching, with the resultant recording of a null file section on another volume. If csw is "0"b, tape_ansi_file_cntl_$data_eot is called to switch volumes. If the returned code is zero, volume switching has occurred and the procedure returns. If the code is nonzero, the volume switch did not take place. This can be due either to an error or to the lack of another volume. If the record format is S or SB and the entire record has not yet been written (left ^= 0), control passes to the error exit. If the record format is other than S or SB, or the entire spanned record has been written (left = 0), this particular iox_$write_record call can complete successfully. Further calls must, however, be inhibited, therefore cseg.lrec.code is set to the error code value. The return code is set to zero, the file is unlocked, and the procedure returns.

Entry: tape_ansi_lrec_io_$close

This entry point is called by tape_ansi_file_cntl_$close to terminate logical record I/O in a consistent manner at close time.

Usage

```
     dcl  tape_ansi_lrec_io_$close entry (ptr, fixed  bin  (35));

     call tape_ansi_lrec_io_$close (acP, code);
```

where:

1.	acP	is a pointer to the cseg. (Input)
2.	code	is a standard status code. (Output)

INTERNAL LOGIC

The cseg pointer is copied from the argument list and the close entry switch (csw) is set to "1"b. This switch governs the action taken if write_buf must be called and either an error or EOT occurs. If the I/O switch is open for sequential_input and the I/O buffer pointer is null, control passes to the buffer management reset exit to perform that function and return. If the pointer is nonnull, control passes to the buffer release exit to perform that function, reset the buffer management strategy, and return.

If the I/O switch is open for sequential_output and there is no current I/O buffer, control passes to the buffer management reset exit. If there is an I/O buffer (cseg.lrec.bufP ^= null) but cseg.offset is either 0 or fd.bo, the I/O buffer does not contain any data. In this case, control passes to the buffer management reset exit. If, however, the processing offset is neither 0 nor fd.bo, the buffer contains data that must be written. In this case, write_buf is called and control passes to the buffer release exit.

The buffer release exit calls tape_ansi_tape_io_$release_buffer to release the current I/O buffer, calls tape_ansi_tape_io_$close to reset the buffer management strategy, and returns. The buffer management reset exit calls tape_ansi_tape_io_$close and returns.

MODULE: tape_ansi_ibm_lrec_io_

This module performs the iox_$read_record and iox_$write_record functions for IBM file sets.

Entry: tape_ansi_ibm_lrec_io_$read_record

This entry point performs the iox_$read_record function.

Usage

```
     dcl  tape_ansi_ibm_lrec_io_$read_record ext entry (ptr, ptr,
          fixed bin (21), fixed bin (21), fixed bin (35));

     call tape_ansi_ibm_lrec_io_$read_record (iocbP, ubP,
          buf_len, rec_len, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	ubP	is a pointer to the user's record buffer. (Input)
3.	buf_len	is the number of characters to be read. (Input)
4.	rec_len	is the number of characters actually read. (Output)
5.	code	is a standard status code. (Output)

The following is a nonexhaustive list of error_table_ codes that can be returned:

file_busy	file in use for other I/O activity; no data returned.
fatal_error	unrecoverable error occurred; all, some, or no data returned. Data can be incorrect.
long_record	actual record length exceeded buf_len (requested length); buf_len characters returned, remainder of record discarded.
invalid_record_desc	a variable-length or spanned record's RDW or SDW is invalid; some or no data returned. Data can be incorrect.

tape_error	a parity error occurred while reading; all, some, or no data returned. Data can be incorrect.

It is important to note that for the blocked record formats, tape_error is returned with the first record of the block that contains the error. Since a parity error is associated with a physical block as opposed to a logical record, the first record may or may not contain the invalid character or characters. If subsequent iox_$read_record calls are made, records from the same block can contain the invalid data even though their return codes are zero.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB and cseg.file_lock is checked to ensure that the file is not in use. If the file is in use, the status code error_table_$file_busy is returned. If the file is not in use, a cleanup handler is established and the file is locked. If invoked, the cleanup handler unlocks the file lock (cseg.file_lock) and sets the logical record I/O lock (cseg.lrec.code) to error_table_$fatal_error. This step is necessary because an interrupted logical I/O operation can leave the internal I/O buffers and logical record processing variables in an inconsistent state. The logical record I/O lock is then checked, and if it is nonzero, the procedure immediately returns that error code.

The desired record can have already been read as the result of an iox_$read_length call. If so, the read_length buffer count will contain a valid value (cseg.rlN ^= -1). If the user's request (buf_len) is equal to or greater than the number of characters in the buffer (cseg.rlN), cseg.rlN characters are returned with status code zero. If buf_len is less than cseg.rlN, buf_len characters are returned with the status code error_table_$long_record. The appropriate number of characters are moved into the user's buffer from the read_length buffer and rec_len is set to the number of characters moved. The variable cseg.rlN is set to -1 to indicate that the read_length buffer no longer contains a valid record. The logical record count (cseg.lrec.reccnt) is incremented, the file is unlocked, and the procedure returns.

If the record is not in the read_length buffer, control is transferred to one of the four format routines. Three automatic variables are used by all four routines to control their operation. The variable remain is set by the internal procedure get_record and contains the number of characters in the block that remain to be processed. The variable move is set by the format routines to the number of characters moved to the user's buffer by the internal procedure move_to_user. The variable req_off is set to the number of characters processed by a single logical record request. It can differ from move and is used by the internal procedure read_release both to locate the beginning of the next record within a block and to release the I/O buffer when the block is exhausted.

For U format, get_record is called to obtain a record. The variable move is set equal to remain, because a U format record fills an entire block. If buf_len is less than move, the long record switch (the automatic variable long_record) is set to "1"b and move is set equal to buf_len, so that the number of characters returned will be the number requested. (In the absence of any other I/O error or event, long_record = "1"b at exit time causes the procedure to return the status code error_table_$long_record.) Since each logical record request requires a new block, req_off is set equal to remain so that the I/O buffer will be released. The procedure move_to_user is then called to move the record to the user's buffer and read_release is called to release the I/O buffer. Control then passes to the normal exit routine.

For F and FB format, get_record is called to obtain a record. If the file's record length (fd.reclen) exceeds the value of remain, a short record situation exists. Since iox_ does not treat this case as an error, move is set equal to remain without setting a status code. If fd.reclen is less than or equal to remain, move is set equal to fd.reclen so that only one record's worth of data is moved. If buf_len is less than move, the user's buffer is too small to contain all the available data. The long record switch is therefore set to "1"b and move is set to buf_len so that only the requested number of characters is moved. The variable req_off is set equal to fd.reclen because each logical record request must process an entire record, even if only a portion of that record is actually moved to the user's buffer. The procedure move_to_user is called to move the data and read_release is called to position beyond the record. (In F format, the I/O buffer is released after every record is processed. In FB format, it is only released after the last record in a block has been processed.) Control then passes to the normal exit routine.

For V and VB format, get_record is called to obtain a record. A pointer to the record's RDW (record descriptor word) is made. The record length is extracted, decremented by 4 (the length of the RDW itself), and set into the automatic variable data_len. The RDW length field is a 15 bit signed binary number (16 bits in all), recorded as two 8-bit frames. When reading in 9-mode, each frame is stored into a 9-bit byte with the high-order bit of each byte set to 0. In order to recompose the original binary number, the low-order 8 bits of the high-order byte must be shifted right by 1 bit, into the high-order bit location of the low-order byte.

If the data length cannot be extracted, or if it exceeds the number of characters remaining in the block (remain - 4), control passes to the invalid record descriptor error exit. If the RDW is valid, move is set equal to data_len. If, however, buf_len is less than move, long_record is set to "1"b and move is reset equal to buf_len. The variable cseg.lrec.offset, the current processing offset within the I/O buffer, is incremented by 4 (the length of an RDW) so that the RDW is not processed as part of the data. The variable req_off is set equal to data_len so that the request will process the entire record, even if only a portion is actually being returned, and move_to_user is called. The procedure read_release is invoked to position beyond the record. Control then passes to the normal exit routine.

For VS and VBS format, get_record is called to obtain a record segment. To keep track of the number of characters that have yet to be moved into the user's buffer to satisfy the request, the automatic variable left is initialized equal to buf_len. The procedure process_sw is invoked to process and validate the segment's SDW (segment descriptor word) and to extract the segment's data length into data_len. If left is greater than or equal to data_len, all the data in the segment is needed and move is set equal to data_len. If left is less than data_len, only a portion of the segment is needed to satisfy the (balance of the) request. In this case, long_record is set to "1"b and move is set equal to left. The procedure move_to_user is invoked to move the data to the user's buffer, left is decremented by the value of move to give the number of characters still required to complete the request, and read_release is called to position beyond the record segment.

The SDW type code is checked. If the code indicates either a complete or terminal record segment, the entire logical record has been processed and control passes to the normal exit routine. If not, the remaining record segments must either be skipped (if the user's request is satisfied), or processed (if their data is needed to complete the request). In the latter case, left is nonzero. The procedure get_record is called to obtain the next record segment and control is passed back to the process_sw call described above. In the former case, left is zero. The procedure skip_segments is called to position beyond the last segment of the record. Control then passes to the normal exit routine with long_record set to "1"b because the user requested fewer characters than the record contained.

The normal exit routine increments the logical record count (cseg.lrec.reccnt). If a parity error has occurred, the return code is set to error_table_$tape_error. Otherwise, it is set to zero or whatever error code has been set by a previous step. If no error has occurred but long_record is "1"b, the return code is set to error_table_$long_record. (An error code therefore overrides the reporting of the long record condition.) The variable rec_len is set equal to the automatic variable total, whose value has been maintained by move_to_user to be the total number of characters placed in the user's buffer. The variable cseg.file_lock is set to "0"b and the procedure returns. The error and invalid record descriptor exit routines perform the same functions as described above, with the exception of incrementing cseg.lrec.reccnt.

INTERNAL PROCEDURES

Entry: get_record

This procedure makes a logical record available to the record format routines, either by reading a new block into an I/O buffer, or by setting the buffer processing variables for the next record already in a buffer.

If the I/O buffer pointer (cseg.lrec.bufP) is nonnull, at least one record is already in the I/O buffer. The variable remain is set to the number of characters not yet processed, and the procedure returns. If cseg.lrec.bufP is null, tape_ansi_tape_io_$read is called to read a block. The variable cseg.lrec.bufP is set to point to the I/O buffer and cseg.lrec.nc_buf is set to the number of characters read. If the call returns a nonzero code, there are two main possibilities.

If the code is error_table_$eof_record, an end-of-file mark has been read. Either tape_ansi_nl_file_cntl_$data_eof or tape_ansi_file_cntl_$data_eof is invoked to determine whether an end-of-file mark has been read or the file is continued on another volume. If a zero code is returned, the file is continued on the next volume. Since tape_ansi_file_cntl_ has performed all necessary volume switching functions, control is simply passed back to the tape_ansi_tape_io_$read call. If the code is nonzero, either no more data exists or an error has occurred, and control passes to the error exit routine.

If tape_ansi_tape_io_$read returns any other nonzero code, a parity or fatal error has occurred. If the code is error_table_$tape_error, the parity error switch (the automatic variable parity_error) is set to "1"b and the current iox_$read_record operation is completed. Any other error code causes control to pass immediately to the error exit routine. (The iox_$control operation "reset_error_lock" can be used to permit further iox_$read_record calls, if and only if the lock value is error_table_$tape_error.)

If the code was zero or error_table_$tape_error, the block count (cseg.lrec.blkcnt) is incremented. If cseg.nc_buf exceeds fd.blklen, cseg.nc_buf is set equal to fd.blklen to eliminate the unwanted characters. There are three possibilities if cseg.nc_buf exceeds fd.blklen. The user can have specified an incorrect block length, causing data to be lost, the block length can be specified with the intent of causing the latter portion of a block to be ignored, or the "extra" characters can have been appended by the MTS500 tape subsystem. The latter case occurs when a block whose length is not evenly divisible by 4 is read, and the subsystem pads the block to a word boundary with octal 000. This behavior is a result of the inability of the current tape device interface (tdcm_) to process blocks on a per-character basis, and should no longer occur when tape_ioi_ becomes the device interface.

Since V, VB, VVS, and VBS format blocks contain BDWs (block descriptor words), the BDW block length field is checked against cseg.nc_buf. If cseg.nc_buf is less than the BDW length value, control passes to the invalid descriptor error exit. (Before performing this comparison, the BDW length field must be recomposed in the same manner as the RDW length field, described above.) The variable cseg.nc_buf is set to the BDW length value, to discard any MTS500 block pad characters not eliminated in the previous fd.blklen check, and cseg.offset is set to 4 (the length of the BDW itself) to indicate that the BDW has been processed. The variable remain is set to the number of characters available for processing (cseg.nc_buf - cseg.offset), and the procedure returns. For U, F, and FB format, cseg.offset is set to 0, remain is set, and the procedure returns.

<u>Entry</u>: process_sw

This procedure is called by the VS and VBS format routine to validate and process an SDW. The SDW length value is recomposed as described above, decremented by 4 (the length of the SDW itself), and the resulting segment data length is stored into data_len. The variable data_len is checked against the actual number of characters remaining in the block and the SDW type code is validated. An inconsistency detected by these checks results in a nonlocal transfer to the invalid record descriptor error exit. If the SDW is valid, cseg.offset is incremented by 4 (the length of an SDW) so that the SDW is not processed as part of the segment's data. The variable req_off is set equal to data_len and the procedure returns.

Entry: skip_segments

This procedure is called by the VS and VBS format routine to skip record segments that are not required to satisfy the user's request (buf_len < total record length). The procedure get_record is called to obtain a record segment and process_sw is called to process and validate its SDW. If the type code is that of a final segment, read_release is called to position beyond it, and the procedure returns. Otherwise, read_release is called and control passes back to the get_record call. This algorithm is continued until the final segment is encountered and skipped.

Entry: move_to_user

This procedure is called by all four format routines to move data from the I/O buffer to the user's buffer. If move is zero, no data is to be moved and the procedure returns. Otherwise, a pointer is made to the first character to be moved from the I/O buffer, and another pointer is made to the location within the user's buffer where that character is to be placed. If the encoding mode (fd.mode) is not EBCDIC, no character conversion need be performed and the data is simply moved. Otherwise, ebcdic_to_ascii_ is invoked to translate and move the data. The automatic variable total is incremented by the value of move, to maintain a count of the total number of characters moved.

Entry: read_release

This procedure is called to release a logical record or record segment from an I/O buffer. If the record is the last (or only) one in the buffer, the entire buffer is also released. The variable cseg.offset (the current processing offset within the buffer) is incremented by the value of req_off (the number of characters processed by the format routine). The value of remain is calculated by subtracting the new current offset from the buffer character count (cseg.nc_buf). (The variable remain can become negative.) If remain is 4 or greater, the procedure returns without releasing the I/O buffer, because the remaining characters must be valid data.

If remain is less than 4, usually the I/O buffer is exhausted and is to be released by calling tape_ansi_tape_io_$release_buffer before the procedure returns. For U format this is always the case, because a U format request always processes every character (req_off is set equal to remain). For V, VB, VVS, and VBVS, this is similarly the case, because fewer than 4 characters does not even allow for a 4-character BDW. For F and FB format however, the buffer is only released if the logical record length is greater than the value of remain. If the logical record length is less than or equal to remain, the procedure simply returns. This causes the loss of short records (short record length < record length < 4) in some unusual, but possible, cases. Unfortunately, this is the only way to avoid processing the pad bytes (octal value 000) appended to blocks that have lengths not evenly divisible by 4.

<u>Entry</u>: tape_ansi_ibm_lrec_io_$write_record

This entry point performs the iox_$write_record function for IBM file sets.

<u>Usage</u>

```
     dcl  tape_ansi_ibm_lrec_io_$write_record entry (ptr,
          ptr, fixed bin (21), fixed bin (35));

     call tape_ansi_ibm_lrec_io_$write_record (iocbP,
          ubP, buf_len, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	ubP	is a pointer to the user's buffer. (Input)
3.	buf_len	is the number of characters to be written. (Input)
4.	code	is a standard status code. (Output)

The following is a nonexhaustive list of error_table_ codes that can be returned.

file_busy	file in use for other I/O activity; record not written.
fatal_error	unrecoverable error occurred; see "Write Errors" below.
long_record	buf_len exceeds the maximum record and/or block length; the record is not written.

eov_on_write	no more records can be written on the current volume. For VS and VBS format, the record can be partially written; for all other formats, the record is not written.
tape_error	a parity I/O error has occurred; see Write Errors below.

WRITE ERRORS

In the case of a fatal or parity error, more records can be affected than just the particular record being written when the error code is returned. Such an error is detected upon the writing of a block, and each iox_$write_record call does not necessarily cause a block to be written. Hence, a zero status code does not guarantee that a record has been written at all, let alone written correctly. FB format blocks always, and VB and VBS blocks can, contain multiple records, so that an error in writing a block affects every record packed into the block.

In addition, tape_ansi_ does not wait to check the status of a write operation after the operation has been issued. Indeed, in the time between issuing a write operation and receiving its status, a number of further write operations can have been issued. This method of operation is termed asynchronous processing and normally is highly satisfactory. If an I/O error occurs, however, not only is the erroneous block not written, but all blocks queued for writing subsequent to the error block are not written. Even in this case it is possible to maintain an accurate block count, but since the number of records per block can vary, an accurate record count cannot be maintained. Since U, F, and V format place only one record per block, the actual number of records written equals the block count (available by calling iox_$control "file_status" operation). For FB, VB, and VBS format, an indeterminate number of records packed into blocks subsequent to the error block are not written, and for VS and VBS format, the record can have been partially written in blocks prior to the error block.

INTERNAL LOGIC

The cseg pointer is obtained from the IOCB and cseg.file_lock is checked to be sure that the file is not busy for other I/O activity. If it is busy, the procedure immediately returns the error code error_table_$file_busy. Otherwise, a cleanup handler is established and the file lock is locked. If invoked, the cleanup handler unlocks the file lock and sets the logical record I/O lock to error_table_$fatal_error. This step is necessary because an interrupted logical I/O operation can leave the I/O buffer and its processing variables in an inconsistent state.

The logical record I/O lock (cseg.lrec.code) is checked to ensure that I/O has not been inhibited due to an unrecoverable error. If cseg.lrec.code is nonzero, the file lock (cseg.file_lock) is unlocked and the procedure returns with code set to the logical I/O lock value. If the file is nonlabeled, the intrafile position indicator (vl (fl.flX).pos) is checked to determine whether or not the tape is positioned in the data portion of the file. The first time iox_$write_record is called subsequent to an opening, the tape is positioned in the file's header label group. In this case, tape_ansi_file_cntl_$position_for_output is called to write the header label group tape mark, such action defining the transition into the data portion of the file. If an error occurs while writing this tape mark, control passes to the error exit routine.

The tape mark is not written until the first logical record write call for the following reason. Volume switching is performed if end-of-tape is detected while writing the header label group. This causes a null file section to be recorded on the old volume. Yet if the I/O switch is closed without an intervening write operation, another null file section is written on the new volume. By inhibiting the detection of end-of-tape until the header label group tape mark is written, and by delaying writing the tape mark until the first write operation, volume switching can be avoided if no write operations are issued. The close call causes just a single null file section to be written on the current volume.

Control then passes to one of the four record format routines. For U format, buf_len is checked to ensure that it does not exceed the maximum number of characters that can fit in a block (fd.blklen). If its value is too large, control passes to the long record error exit. The procedure get_buf is called to obtain an I/O buffer and move (the number of characters to be moved from the users buffer) is set equal to buf_len. The variable req_off (the number of characters to be written by this request) is also set equal to buf_len and move_to_buf is called to move the user's data into the I/O buffer. The procedure write_buf is called to write the block, and control passes to the normal exit routine.

For F and FB format, buf_len is checked to ensure that it does not exceed the record length (fd.reclen). If it does, control passes to the long record error exit routine. The procedure get_buf is called to obtain an I/O buffer, if necessary. (For F format, an I/O buffer is obtained for each call, because each record requires a new block.) The variable move is then set equal to buf_len and remain is set to the number of pad characters that must be appended to the user's data to make a complete record (fd.reclen - buf_len). This step is necessary because fixed-format records must be of identical lengths. If remain is nonzero, the appropriate number of blanks are inserted into the I/O buffer. The variable req_off is set equal to fd.reclen because each request processes a complete record, even if buf_len is less than fd.reclen. The procedure move_to_buf is called to move the user's data into the I/O buffer immediately before the inserted padding (if any). If records are not blocked (F format), write_buf is called to write the record. Otherwise, write_buf is not called unless the block contains as many records as can fit (cseg.offset = fs.blklen). Control then passes to the normal exit routine.

For V and VB format, data_len (the length of the record) is set to buf_len plus 4 (the length of an RDW). The variable data_len is checked to ensure that it does not exceed fd.reclen. If it does, control passes to the long record error exit. The procedure get_buf is called to obtain an I/O buffer, if necessary. (An I/O buffer is always obtained for V format, since each record requires a new block.) For VB format, it must be determined if the record to be written can fit into the current block, or if a new block is required. If data_len exceeds the number of remaining characters in the block (fd.blklen - cseg.offset), write_buf is called to write the current block and get_buf is called to obtain a new I/O buffer. Either way, a pointer is made to the I/O buffer location where the record's RDW is to be constructed.

The RDW location is saved in cseg.saveP. Because of the aforementioned tdcm_ - MTS500 block length problems, the length of blocks being written must be evenly divisible by 4 to avoid octal 000 padding out to the word boundary. Such padding would not be reflected in the block's BDW and would cause the block to be unreadable by an IBM system. To avoid this problem, the last record of a V or VB format block is extended with blanks out to the word boundary and the BDW is adjusted accordingly. Of course, the RDW for the extended record must be similarly incremented. Its location is saved for this reason.

The record length (data_len) is decomposed and placed into the RDW length field. (The decomposition process is the reverse of the RDW recomposition process described above, done for the same reason.) The variable cseg.offset is incremented by 4, so that the RDW is considered when computing the total block length, and req_off is set equal to buf_len. The variable move is also set equal to buf_len and move_to_buf is called to move the user's data. If records are not blocked, write_buf is called to write the record. Otherwise, write_buf is not called unless another record could not fit in the current block (fd.blklen - cseg.offset < 4, where 4 is the length of a zero-length record). Control then passes to the normal exit routine.

For VS and VBS format, buf_len is checked to ensure that it does not exceed fd.reclen. If it does, control passes to the longer record error exit. The procedure get_buf is called to obtain an I/O buffer, if necessary. (An I/O buffer is always obtained for VS format, because each record segment requires a new block.) The variable left contains the number of characters still to be moved from the user's buffer, and is initialized equal to buf_len. As each record segment is written, left is decremented by the number of characters written in that segment. The variable remain is set to the number of characters remaining in the current block (fd.blklen - cseg.offset). A pointer is made to the I/O buffer location where an SDW is to be constructed, and the location is saved in cseg.saveP for the reason described above.

The number of characters that can still be placed into the current block is computed. If left, the (balance of the) user's request, plus 4 (the length of an SDW) characters is greater than remain, move is set to as many data characters as will fit (remain - 4). If left + 4 is not greater than remain, the (balance of the) user's request can fit entirely within the current block. It must then be determined whether or not sufficient characters would remain in the block to contain a segment of a subsequent record. If left + 4 is less than or equal to remain - 5, sufficient room would remain for a 5 character segment (4 character SDW plus 1 data character) of the next record. In this case, move is set equal to left and the (balance of the) user's request is placed into the current block.

If, however, a segment of a subsequent record could not fit into the current block, the current segment of the current record is the last segment in the block. Steps must be taken to ensure that placing the block segment into the block does not result in a block with a length not evenly divisible by 4. Such a block would be padded with octal 000 out to a word boundary, resulting in unreadable blocks, as described above. The number of characters of the segment that would be placed into the last word of the block is computed. If the word would be filled, move is set equal to left because no padding occurs. Otherwise, move is set equal to left decremented by the number of characters that would be placed in the last word. Those characters are written in a subsequent segment in the next block. The variable left is then decremented by the value of move to give the amount of data (if any) to be written in subsequent segments.

The type code set in the SDW is dependent upon both the amount of data still to be written (left), and whether or not the segment to be written is the first of the record. If no data from the record has been previously placed into another segment (first_scan = "1"b) and no data remains to be written in a subsequent segment (left = 0), the type is complete; i.e., the segment contains the entire record. If some data has been placed into another segment (first_span = "0"b), and no data remains to be written, the type is final; i.e., the segment is the last of a group of segments that <u>in</u> <u>toto</u> make up the record. If no data from the record has been previously placed into another segment and more remains to be written in subsequent segments (left ^= 0), the type is initial; i.e., the segment is the first of a group of segments that <u>in</u> <u>toto</u> make up the record. If some data has been placed into another segment and more remains to be written, the type is medial; i.e., the segment is one of a group of three or more segments (but neither the first nor the last) that <u>in</u> <u>toto</u> make up the record.

The variable data_len, the actual record segment length, is set equal to move + 4 (to include the SDW), decomposed (as described above), and placed into the SDW length field. For DOS files (cseg.standard = 3), a special check is made for zero-length record SDWs. If the SDW length value is 4 (no data), the high-order bit of the SDW length field must be set to "1"b. The variable cseg.offset is incremented by 5, so that the SDW is considered when computing the total block length. The variable req_off is set equal to move, and move_to_buf is called to move the user's data into the I/O buffer. The variable remain is set to the number of characters now The variable remaining in the block (remain - data_len). If record segments are not blocked (VS format), control passes to write the I/O buffer. If record segments are blocked, the I/O buffer is only written if another nonzero length record segment could not fit into the block (remain < 5).

If the I/O buffer is to be written, write_buf is called to write it, get_buf is called to obtain another, and remain is set to the number of characters available in the new buffer (fd.blklen - cseg.offset). Whether or not the I/O buffer was written, left is checked to determine whether or not the user's request has been satisfied. If left is nonzero, control passes back to make another SDW pointer for the next record segment; if left is zero, control passes to the normal exit routine.

The normal exit routine increments the logical record count (cseg.lrec.reccnt) and sets the return code and cseg.lrec.code to the code returned by the last I/O operation (normally zero). If csw (the tape_ansi_ibm_lrec_io_$close entry switch) is "1"b, control then passes to that entry's exit routine. Otherwise, the file lock is unlocked and the procedure returns. (The variable csw is initialized to "0"b upon procedure block activation, but is set to "1"b by the tape_ansi_ibm_lrec_io_$close entry.)

The long record and error exits perform similar functions, with the exception of incrementing cseg.lrec.reccnt.

Entry: get_buf

This procedure is called to obtain an I/O buffer, if one is needed. If the I/O buffer pointer (cseg.lrec.bufP) is nonnull, a buffer is available and the procedure returns. If it is null, tape_ansi_tape_io_$get_buffer is called to make an I/O buffer available. If the record format is V, VB, VVS, or VBVS, the current offset within the buffer (cseg.offset) is set to 4, to reserve space for the BDW. For all other formats, it is set to 0.

Entry: move_to_buf

This procedure is called to move data from the user's buffer to the I/O buffer. If move is zero, there is no data to be moved. In this case, cseg.offset is incremented by the value of req_off (the number of characters processed by the request) and the procedure returns. (The variable cseg.offset must be incremented to allow for the case of zero-length records in V, VB, VVS, and VBS format. Such records consist of RDWs or SDWs alone.) If data is to be moved, pointers are made to the offset within the I/O buffer where the data is to be placed, and to the offset in the user's buffer from which the data is to be taken. If the encoding mode (fd.mode) is ASCII, the data is moved. If the mode is EBCDIC, ascii_to_ebcdic_ is called to translate and move the data. The variable total is incremented by the value of move, to maintain a count of the total number of characters moved. The variable cseg.offset is incremented by req_off, and the procedure returns.

Entry: write_buf

This procedure writes a block, appending block pad characters if necessary. If cseg.offset is less than 20 and the format is neither F nor FB, the block must be padded. This step is necessary for two reasons: 1) blocks of fewer than 18 characters must not be written, and 2) blocks to be written must consist of an integral number of words (4 characters/word). The number of pad characters is computed by subtracting cseg.offset (the number of characters presently in the block) from 20, and control passes to perform the padding.

If cseg.offset is greater than or equal to 20 but not evenly divisible by four, the block must still be padded to satisfy requirement 2) above. Padding for both cases is performed by inserting the appropriate number of pad characters into the I/O buffer immediately following its current contents, and then incrementing cseg.offset to reflect the new, adjusted block length. In addition, for V, VB, VS, and VBS format, the BDW and last RDW in the block must be incremented to reflect the addition of the padding.

The entry point tape_ansi_tape_io_$write is called to write the block. If the return code is zero, the block count (cseg.lrec.blkcnt) is incremented and the procedure returns. If the return code is nonzero, there are two major possibilities. If the code is not error_table_$eov_on_write, an error has occurred, and the block count is decremented if more than one block was not written (cseg.blkcnt = cseg.blkcnt - cseg.soft_status.nbuf + 1). (The suspended buffer count is currently obtained directly from the cseg. Eventually, when tape_ioi_ becomes the device interface, a tape_ioi_ status entry will be called to obtain this value.) The logical record count is invalidated by setting it negative, and control passes to the error exit.

If the code is error_table_$eov_on_write, then end-of-tape has been detected. This is more in the nature of an event than an error. The variable cseg.lrec.blkcnt is incremented, because the block has been successfully written. If csw is "1"b (i.e., the procedure was entered at the $close entry point), the procedure simply returns. This is done so that EOT detection at close time does not force volume switching, with the resultant recording of a null file section on another volume. If csw is "0"b, either tape_ansi_nl_file_cntl_$data_eot or tape_ansi_file_cntl_$data_eot is called to switch volumes. If the returned code is zero, volume switching has occurred and the procedure returns. If the code is nonzero, the volume switch did not take place. This can be due either to an error or to the lack of another volume. If the format is VS or VBS and the entire record has not yet been written (left ^= 0), control passes to the error exit. If the format is other than VS or VBS, or if VS or VBS and the entire record has been written (left =

0), this iox_$write_record operation is not in error. Further operations must be inhibited nevertheless, because there is no more room on the volume. To this end, cseg.lrec.code (the logical I/O lock) is set to the error code value. The return code is then set to 0, because this operation is successful, and control passes to unlock the file lock and return.

Entry: tape_ansi_ibm_lrec_io_$close

This entry point is called by tape_ansi_file_cntl_$close to terminate logical record I/O in a consistent manner at close time.

Usage

```
     dcl  tape_ansi_ibm_lrec_io_$close entry (ptr,
          fixed bin (35));

     call tape_ansi_ibm_lrec_io_$close (acP, code);
```

where:

1.	acP	is a pointer to the cseg. (Input)
2.	code	is a standard status code. (Output)

INTERNAL LOGIC

The cseg pointer is copied from the argument list and the close entry switch (csw) is set to "1"b. This switch governs the action taken if write_buf must be called and either an error or EOT occurs. If the I/O switch is open for sequential_input, and the I/O buffer pointer is null, control passes to the buffer management reset exit to perform that function and return. If the pointer is nonnull, control passes to the buffer release exit to perform that function, reset the buffer management strategy, and return.

If the I/O switch is open for sequential_output and there is no current I/O buffer, control passes to the buffer management reset exit. Even if an I/O buffer exists, it may not contain any valid data. If cseg.offset is zero, it surely does not, and if cseg.offset is 4 and the record format is V, VB, VS, or VBS, the buffer only contains a BDW. In either case, control passes to the buffer management reset exit. Otherwise, the buffer contains data that must be written. The procedure write_buf is called, and control passes to the buffer release exit.

The buffer release exit calls tape_ansi_tape_io_$release_buffer to release the current I/O buffer, calls tape_ansi_tape_io_$close to reset the buffer management strategy, and returns. The buffer management reset exit calls tape_ansi_tape_io_$close and returns.

MODULE: tape_ansi_read_length_

This module performs the iox_$read_length function. It reads a record, returns its length, and saves the record in a buffer for future use by an iox_$read_record call.

Usage

```
     dcl  tape_ansi_read_length_ entry (ptr, fixed bin (21),
          fixed bin (35));

     call tape_ansi_read_length_ (iocbP, reclen, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	reclen	is the length of the next record, in characters. (Output)
3.	code	is a standard status code. (Output)

If code is error_table_$tape_error, the record length is returned but can be in error. If code is any other nonzero value, the record length is undefined.

Internal Logic

The cseg pointer is obtained from the IOCB and cseg.invalid is checked to determine if the cseg has an internal inconsistency. If it does, the procedure immediately returns the error code error_table_$invalid_cseg. The variable cseg.file_lock is checked to ensure that the file is not in use for other I/O activity. If it is in use, the procedure immediately returns the error code error_table_$file_busy.

Otherwise, a cleanup handler is established and the file is locked. If invoked, the cleanup handler unlocks the file lock and sets the logical record I/O lock (cseg.lrec.code) to error_table_$fatal_error. This action is necessary because an interrupted read_length operation can leave the logical record processing variables in an inconsistent state.

The read_length buffer pointer (cseg.rlP) is checked to determine whether or not a read_length buffer exists. If the pointer is null, one does not. The external procedure hcs_$make_seg is invoked to make a segment in the process directory. The entry name of the segment is formed as follows:

```
module_name || first_volname || "_.rl"
```

where module_name is the name of the I/O module (tape_ansi_ or tape_ibm_) and first_volname is the volume name of the first (or only) volume of the volume set. If an error occurs while making the segment, the procedure returns the code error_table_$fatal_error. If no error occurs, the maximum buffer length is computed and saved in the internal static variable nc_wanted. When reading a record to determine its length, the procedure must be sure to request every possible character in the record, and no record can contain more than nc_wanted characters. Control then passes to read a record.

If cseg.rlP is nonnull, the read_length buffer already exists. The buffer character count (cseg.rlN) is checked to determine whether or not the buffer already contains a record. This is possible if two iox_$read_length calls are issued without an intervening iox_$read_record call; the second iox_$read_length call references the same record as the first. If cseg.rlN is not equal to -1, the buffer already contains a record. The return code is set to zero, reclen is set equal to cseg.rlN, the file is unlocked, and the procedure returns.

If cseg.rlN is -1, then a record must be read into the buffer. To do so, the file must first be unlocked, and tape_ansi_lrec_io_$read_record or tape_ansi_ibm_lrec_io_$read_record must be called to read the record. The call requests nc_wanted characters. The variable cseg.rlN is set to the number actually read. The file is then locked once again. If the returned code is either zero or error_table_$tape_error, the logical record count (cseg.lrec.reccnt) is decremented. This is done because although the record count was incremented by the read_record call, the record has not actually been read (by the user). The variable reclen is set to cseg.rlN, the file is unlocked, and the procedure returns.

If the returned code is any other value, the read_record operation has failed. The variable reclen is set to zero, cseg.rlN is set to -1 (to ensure that the buffer contents are invalidated), the file is unlocked, and the procedure returns.

MODULE: tape_ansi_position_

This module implements the iox_$position function. Positioning to beginning-of-file, end-of-file, and forward a specified number of records are supported. Positioning backwards a specified number of records is not supported.

Usage

```
     dcl  tape_ansi_position_ entry (ptr, fixed bin, fixed bin,
          fixed bin (35));

     call tape_ansi_position_ (iocbP, type, n, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	type	specifies the type of positioning to be performed. The following types are supported: -1 beginning-of-file 0 forward n records (see n below) 1 end-of-file (Input)
3.	n	specifies the number of records to be positioned over, if type = 0. If type ^= 0, n is ignored. The value of n must be $\geq$ 0. If n = 0, no action is performed. (Input)
4.	code	is a standard status code. (Output)

The following is a nonexhaustive list of error_table_ codes that can be returned.

invalid_cseg	the control segment is invalid; the operation was not performed.
file_busy	the file is already in use for other I/O activity; the operation was not performed.
fatal_error	an unrecoverable I/O error occurred; the operation may or may not have been completed. The I/O switch may or may not have been closed.
tape_error	a parity I/O error occurred. If the I/O switch is open, the operation was completed. If not, the operation may not have been completed.

end_of_info	logical end-of-file encountered before completing a position forward n records request. The file is positioned at end-of-file.
bad_arg	either type or n is invalid; the operation was not performed.

Internal Logic

The cseg pointer is extracted from the IOCB. The variable cseg.invalid is checked to determine whether or not the control segment is valid. If it is not valid, the error code error_table_$invalid_cseg is returned. Otherwise, the file lock (cseg.file_lock) is checked to determine whether the file is already busy for other I/O activity. If it is busy, the procedure returns the error code error_table_$file_busy. Otherwise, a cleanup handler is established and cseg.file_lock is set. If invoked, the cleanup handler sets the logical record I/O lock to error_table_$fatal_error and unlocks the file lock. This step is necessary because an interrupted positioning operation can leave the logical record processing variables in an inconsistent state.

The type argument is validated to ensure that it falls within the range $-1 \leq$ type $\leq +1$. If it does, control passes to perform the appropriate positioning operation. If it does not, the return code is set to error_table_$bad_arg and control passes to the exit routine.

Position to beginning-of-file

Either tape_ansi_nl_file_cntl_$beginning_of_file or tape_ansi_file_cntl_$beginning_of_file is called to perform the actual positioning operation. If the returned code is nonzero, the logical record I/O lock is set to that value. Control then passes to the exit routine.

Position to end-of-file

Either tape_ansi_nl_file_cntl_$end_of_file or tape_ansi_file_cntl_$end_of_file is called to perform the actual positioning operation. If the returned code is nonzero, the logical record I/O lock is set to that value. Control then passes to the exit routine.

Position forward n records

The return code is initialized to zero because no procedure calls can be made. The automatic variable tape_error is initialized to "0"b. This variable is used to determine whether or not a parity error has occurred in the course of positioning.

If n = 0, no records are to be skipped and control passes to the exit routine. Because each block can contain an indeterminate number of records, it would be necessary to maintain a logical record map for every block to implement positioning backwards. Since the cost of such an implementation is excessive, n < 0 is not supported. If n < 0, the return code is set to error_table_$bad_arg and control passes to the exit routine. If n > 0, the argument is copied into the automatic variable i so that the record count can be decremented without affecting the caller's parameter.

The read_length buffer character count (cseg.rlN) is checked to determine whether or not the buffer contains a record. If it does (cseg.rlN ^= -1), the buffer is "emptied" (cseg.rlN = -1) and the record count is decremented. These actions are logically equivalent to skipping 1 record. If there was no record in the read length buffer or if additional records must be skipped, positioning involves physical tape motion.

Records are skipped by invoking either tape_ansi_lrec_io_$read_record or tape_ansi_ibm_lrec_io_$read_record. The variable cseg.file_lock is set to "0"b so that the logical I/O procedure does not find the file locked upon invocation. The appropriate procedure is called with a null user buffer pointer and a zero buffer length so that a logical record is processed but no information is returned. If the returned code is either zero or error_table_$long_record, the read is considered to have completed normally. (Since the length of the record read is almost always greater than the buffer length (0), code is almost always error_table_$long_record. In this case, code is reset to zero and ignored. If the record read has zero length, code is zero.) If the returned code is error_table_$tape_error, a parity error has occurred. This error does not absolutely preclude further reading. In order to continue, the logical I/O lock is unlocked (cseg.lrec.code = 0) and tape_error is set to "1"b so that the procedure eventually returns error_table_$tape_error to its caller. If the returned code is any other value (error_table_$end_of_info, error_table_$fatal_error, etc.), no further positioning is possible and control passes to the exit routine. If processing is to continue, the file lock is locked again and the above algorithm is repeated until the positioning request has been satisfied.

Once the request is complete, tape_error is checked to determine whether a parity error has occurred while processing. If so, cseg.code is set to error_table_$tape_error, relocking the logical record I/O lock, and the return code is set to error_table_$tape_error.

The exit routine unlocks the file lock (cseg.file_lock) and returns whatever code has been set in a previous step.

MODULE: tape_ansi_mount_cntl_

This procedure performs all the volume and device management functions of the I/O module. Currently coded to use the tdcm_ interface, it must eventually be recoded to use rcp_ and tape_ioi_. The internal logic descriptions are therefore confined to describing the functions performed and ignoring the particulars of implementation.

Entry: tape_ansi_mount_cntl_$mount

This entry point is called to assign a device, mount a volume on that device, and read the volume's VOL1 label (if any).

Usage

```
     dcl  tape_ansi_mount_cntl_$mount entry (ptr, fixed bin,
          fixed bin (35));

     call tape_ansi_mount_cntl_$mount (cP, vlX, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	vlX	is the index of the volume link associated with the volume to be mounted. (Input)
3.	code	is a standard status code. (Output)

If code is nonzero, the volume is not mounted and no device is assigned.

INTERNAL LOGIC

A cleanup handler is established that calls the internal procedure cleaner. If invoked, cleaner demounts the volume (if mounted) and unassigns the device (if assigned). After the cleanup handler is established, a device is assigned and the active drive count (cseg.nactive) is incremented. The internal procedure mount_request is called to issue a mount message to the user, mount the desired volume (specified in the volume link), and issue another message when the mount is complete. The internal procedure VOL1_check is called to validate the VOL1 label against its expected characteristics and set the VOL1 status variable (vl.write_VOL1) accordingly. The volume link is filled with the assignment, mount, and VOL1 validation data, and the procedure returns. If an error occurs during any of the above steps, control passes to the error exit routine.

The error exit routine invokes the internal procedure cleaner, sets the return code to error_table_$bad_mount_request, and returns.

<u>Entry</u>: tape_ansi_mount_cntl_$remount

This entry point is called to demount a volume from an assigned device and mount a different volume on the same device.

<u>Usage</u>

```
     dcl  tape_ansi_mount_cntl_$remount entry (ptr, fixed bin,
          fixed bin, fixed bin (35));

     call tape_ansi_mount_cntl_$remount (cP, down_vlX,
          vlX, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	down_vlX	is the index of the volume link associated with the volume to be demounted. (Input)
3.	vlX	is the index of the volume link associated with the volume to be mounted. (Input)
4.	code	is a standard status code. (Output)

If code is nonzero, the requested volume has not been mounted but the volume to be demounted may have been demounted and its device unassigned.

INTERNAL LOGIC

A cleanup handler is established to call the internal procedure cleaner, described above. The current file position (vl.cflX) of the volume to be demounted is invalidated and the internal procedure unload is called to demount the volume. The volume link's device identifier (vl.rcp_id) is invalidated and control passes to call mount_request, continuing as described above.

Entry: tape_ansi_mount_cntl_$insert_rings

This entry point is called to demount all mounted volumes, request that write permit rings be inserted, and mount the volumes again. The write ring switch (cseg.write_ring) is set to "1"b, indicating that all volumes are to be mounted with write permit rings.

Usage

```
     dcl  tape_ansi_mount_cntl_$insert_rings entry (ptr,
          fixed bin (35));

     call tape_ansi_mount_cntl_$insert_rings (cP, code);
```

Entry: tape_ansi_mount_cntl_$write_protect

This entry point is called to issue a hardware file protect order to every assigned device. The write protect switch (cseg.protect) is set to "1"b, indicating that writing is inhibited.

Usage

```
     dcl  tape_ansi_mount_cntl_$write_protect entry (ptr,
          fixed bin (35));

     call tape_ansi_mount_cntl_$write_protect (cP, code);
```

Entry: tape_ansi_mount_cntl_$write_permit

This entry point is called to issue a hardware file permit order to every assigned device. The write protect switch is set to "0"b, indicating that writing is not inhibited.

Usage

```
     dcl  tape_ansi_mount_cntl_$write_permit entry (ptr,
          fixed bin (35));

     call tape_ansi_mount_cntl_$write_permit (cP, code);
```

Entry: tape_ansi_mount_cntl_$free

This entry point is called to demount a volume and unassign its device.

Usage

```
     dcl  tape_ansi_mount_cntl_$free entry (ptr, fixed bin,
          fixed bin (35));

     call tape_ansi_mount_cntl_$free (cP, vlX, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	vlX	is the index of the volume link associated with the volume to be demounted, and whose device is to be unassigned. (Input)
3.	code	is a standard status code. (Output)

If code is nonzero, the volume may not have been demounted and the device may not have been unassigned.

Internal Logic

A cleanup handler is established, as described above, and the current file position (vl.cflX) is invalidated. The volume is then demounted and its device unassigned. The active drive count (cseg.nactive) is decremented, the volume link's device identifier (vl.rcp_id) is invalidated, and the procedure returns.

If an error occurs during any of the above steps, control passes to the error exit routine.

Internal Procedures

The only internal procedure described is VOL1_check. The others have been functionally described in the above text and are highly dependent in their implementation upon the tdcm_ interface.

Entry: VOL1_check

This internal procedure validates the VOL1 label (if any) of a newly mounted volume and sets the VOL1 status variable (vl.write_VOL1) accordingly. This variable takes the following values:

0	the VOL1 label is correct. For an ANSI file set, this means that the first block is an ANSI VOL1 label. For an IBM file set, this means that the first block is an IBM SL VOL1 label. If a density has been specified or inferred (cseg.density ^= -1), the VOL1 label density meets the specification. In addition, the recorded volume identifier matches the expected volume identifier.
1	the tape is blank; i.e., the first read operation detected 25 feet of blank tape and returned blank-tape-on-read status.
2	the first block is unreadable. Either the volume is recorded at an unreadable density, or with the wrong number of tracks, or the tape is defective, or the hardware is malfunctioning, etc.
3	the first block is not a VOL1 label. (An IBM VOL1 label is not treated as such in the context of an ANSI file set.)
4	the first block is a valid VOL1 label, but the recorded volume identifier does not match the expected volume identifier.

5	the VOL1 label is correct in all respects but density. The recorded density does not meet the specified or inferred density (cseg.density).

Currently, this procedure operates independently of the (eventual) rcp_ volume registration mechanism. It must eventually be modified to work in accordance with that mechanism, The majority of its checking functions will be performed by rcp_ itself.

MODULE: tape_ansi_tape_io_

This procedure performs the actual tape operations required by the I/O module. Currently, the procedure is an interface to tdcm_. When tape_ioi_ is implemented, the I/O module can be recoded to call tape_ioi_ directly, or else this procedure should be rewritten to interface to tape_ioi_. The following documentation provides only a functional description of each entry point, since the implementation is entirely tdcm_ dependent.

Entry: tape_ansi_tape_io_$attach

This entry point is called to initialize the tdcm_ tseg contained in the control segment. Currently, it is called only once at initial attach time, before a device has been attached. Eventually, it should perform the tape_ioi_$initialize function and be called (multiply) at device assignment time.

Usage

```
     dcl  tape_ansi_tape_io_$attach entry (ptr);

     call tape_ansi_tape_io_$attach (cP);
```

where cP is a pointer to the control segment. (Input) (Input)

Entry: tape_ansi_tape_io_$open

This entry point is called at logical record I/O open time (lrec_open internal procedure in tape_ansi_file_cntl_ and tape_ansi_nl_file_cntl_) to initialize the tseg for asynchronous I/O. Eventually, it should call tape_ioi_ to set buffer sizes, I/O modes, etc.

Usage

```
     dcl  tape_ansi_tape_io_$open entry (ptr);

     call tape_ansi_tape_io_$open (cP);
```

where cP is a pointer to the control segment. (Input)

Entry: tape_ansi_tape_io_$close

This entry point is called at logical record I/O close time (by tape_ansi_lrec_io_$close or tape_ansi_ibm_lrec_io_$close) to synchronize the tape, backspacing if necessary in the read case, writing the remaining buffers in the write case.

Usage

```
     dcl   tape_ansi_tape_io_$close  entry (ptr, fixed bin (35));

     call tape_ansi_tape_io_$close (cP, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	code	is a standard status code. (Output)

The value of code can be either zero or error_table_$fatal_error. (EOT detection during write synchronization is ignored.)

Entry: tape_ansi_tape_io_$get_buffer

This entry point is called to obtain a pointer to an I/O buffer that will subsequently be written.

Usage

```
     dcl  tape_ansi_tape_io_$get_buffer entry (ptr, ptr,
          fixed bin (35));

     call tape_ansi_tape_io_$get_buffer (cP, bP, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	bP	is a pointer to the I/O buffer. (Output)
3.	code	is a standard status code. (Output)

The value of code can be either zero or error_table_$fatal_error. In the latter case, bP is null.

Entry: tape_ansi_tape_io_$release_buffer

This entry point is called to release an I/O buffer once it is no longer needed; i.e., subsequent to a read operation or after a get_buffer call if no write is to be issued.

Usage

```
     dcl  tape_ansi_tape_io_$release_buffer entry (ptr, ptr,
          fixed bin (35));

     call tape_ansi_tape_io_$release_buffer (cP, bP, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	bP	is a pointer to the I/O buffer to be released. (Input)
3.	code	is a standard status code. (Output)

The value of code can be either zero or error_table_$fatal_error.

Entry: tape_ansi_tape_io_$read

This entry point is called to read one block in asynchronous mode.

Usage

```
     dcl  tape_ansi_tape_io_$read entry (ptr, ptr, fixed bin,
          fixed bin (35));

     call tape_ansi_tape_io_$read (cP, bP, ccount, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	bP	is a pointer to the I/O buffer containing the block. (Output)
3.	ccount	is the number of characters read. (Output)
4.	code	is a standard status code. (Output)

If code is zero, the block was read correctly. The following error_table_ codes can be returned:

eof_record	an end-of-file mark was read; bP is null and ccount is 0.
blank_tape	25 feet of blank tape read; bP is null and ccount is 0.
tape_error	parity error detected; a block was read.
fatal_error	unrecoverable program or I/O error; bP is null and ccount is 0.

Entry: tape_ansi_tape_io_$sync_read

This entry point is called to read a block in synchronous mode. The block is read into a special synchronous I/O buffer pointed to by cseg.syncP.

Usage

```
     dcl  tape_ansi_tape_io_$sync_read entry (ptr, fixed bin,
          fixed bin (35));

     call tape_ansi_tape_io_$sync_read (cP, ccount, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	ccount	is the number of characters read. (Output)
3.	code	is a standard status code. (Output)

If code is zero, the read was successful. The following error_table_ codes can be returned: eof_record, blank_tape, tape_error, and fatal_error. If code is nonzero, ccount is zero.

Entry: tape_ansi_tape_io_$sync_write

This entry point is called to write a block in synchronous mode. The block is written from a special synchronous I/O buffer pointed to by cseg.syncP.

Usage

```
     dcl  tape_ansi_tape_io_$sync_write entry (ptr, ccount,
          fixed bin (35));

     call tape_ansi_tape_io_$sync_write (cP, ccount, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	ccount	is the number of characters to be written. (Input)
3.	code	is a standard status code. (Output)

If code is zero, the write was successful. The following error_table_ codes can be returned:

eov_on_write	end-of-tape was detected; the block was written correctly.
tape_error	a parity error occurred; the block was not written or was written incorrectly.
fatal_error	an unrecoverable program or I/O error occurred; the block was not written.

Entry: tape_ansi_tape_io_$write

This entry point is called to write a block in asynchronous mode.

Usage

```
     dcl  tape_ansi_tape_io_$write entry (ptr, ptr, fixed bin,
          fixed bin (35));

     call tape_ansi_tape_io_$write (cP, bP, ccount, code);
```

where:

1.	cP	is a pointer to the control segment. (Input)
2.	bP	is a pointer to the I/O buffer to be written. (Input)
3.	ccount	is the number of characters to be written. (Input)
4.	code	is a standard status code.

If code is zero, the block was written correctly. The following error_table_ codes can be returned:

eov_on_write	end-of-tape was detected; the block was written correctly.
tape_error	a parity error occurred; the block was not written.
fatal_error	an unrecoverable program or I/O error occurred; the block was not written or was written incorrectly.

Entry: tape_ansi_tape_io_$order

This entry point is called to issue an order operation. The following orders can be issued:

bsf	backspace file
bsr	backspace record
ers	erase
fsf	forward space file
fsr	forward space record
rqs	request status
rss	reset status

rew	rewind
run	rewind and unload
eof	write end-of-file mark
pro	set file protect
per	set file permit
sdn	set density (qualified further)

Usage

```
     dcl  tape_ansi_tape_io_$order entry (ptr, char (3),
          fixed bin, fixed bin (35));

     call tape_ansi_tape_io_$order (cP, order, q, code);
```

where:

1. cP — is a pointer to the control segment. (Input)

2. order — is the order to be performed, as listed above. (Input)

3. q — is the order qualifier. The value of q is ignored unless the order is "sdn". In this case, q can be:

0	200 bpi
1	556 bpi
2	800 bpi
3	1600 bpi (Input)

4. code — is a standard status code. (Output)

If code is zero, the order was performed correctly. The following error_table_ codes can be returned:

fatal_error	possible for all orders; an unrecoverable program or I/O error occurred. The order may or may not have been performed.
positioned_on_bot	possible for bsf and bsr only; the tape is/was positioned at beginning-of-tape. The order may or may not have been performed.
eov_on_write	possible for ers and eof only; end-of-tape detected. The order was performed correctly.
eof_record	possible for fsr and bsr only; the order spaced over an end-of-file mark.
tape_error	possible for all orders; an I/O error occurred. The order may or may not have been performed.

MODULE: tape_ansi_interpret_status_

This module is called by tape_ansi_tape_io_ to interpret the IOM status bits. It generates an array of error_table_ status codes. When tape_ioi_ becomes the device I/O interface, this module will no longer be needed.

Usage

```
     dcl  tape_ansi_interpret_status_ entry (ptr);

     call tape_ansi_interpret_status_ (hP);
```

where:

1. hP   is a pointer to a hardware status structure. (Input)

The hardware status structure is declared as follows:

```
     dcl 1 hdw_status based (hP),
         2 iom_bits bit (72) aligned,       /* IOM status bits */
         2 no_minor fixed bin,         /* number of minor codes */
         2 major fixed bin (35),           /* major status code */
         2 minor (10) fixed bin (35);     /* minor status codes */
```

Internal Logic

The procedure is passed the structure with hdw_status.iom_bits set to the IOM status to be interpreted. The variable hdw_status.no_minor is set to the number of minor status codes, the major status code is placed in hdw_status.major, and the hdw_status.minor array is filled with the minor status code(s).

MODULE: tape_ansi_parse_options_

This module is called by tape_ansi_attach_ to validate an iox_ attach description.

Usage

```
     dcl  tape_ansi_parse_options_ entry (ptr, (*) char (*)
          varying, char (32) varying, fixed bin (35));

     call  tape_ansi_parse_options_ (taoP, options, error, code);
```

where:

1.	taoP	is a pointer to the attach options structure (tao., as declared by tape_attach_options.incl.pl1. (Input)
2.	options	is an array of attach description lexemes, as parsed by iox_. (Input)
3.	error	is a diagnostic message. It is null if code is zero; it can be nonnull if code is nonzero. (Output)
4.	code	is a standard status code. If code is nonzero, the attach description is invalid. (Output)

Internal Logic

The variables error and code are initialized to "" and zero, respectively. If the number of elements in the options array (tao.noptions) is zero, the procedure immediately returns the error code error_table_$noarg because the attach description cannot be null.

Processing begins with the volume list, which is the first section of the attach description. The array index i is initialized to 1. The variable hyphen_ok is set to "0"b to indicate that the first options array element should be a volume name and therefore should not begin with a hyphen. The element is tested to determine if it is "-volume" or "-vol", either of which indicates that the next element is a volume name that may or may not begin with a hyphen. If the element is "-volume" or "-vol", hyphen_ok is set to "1"b and no_next is invoked to determine whether or not the next element exists. If the next element does not exist, the procedure returns the error code set by no_next because the -volume option requires a following volume name. If the next element exists, no_next has incremented the array index to access it.

The next element is tested to determine whether or not it begins with a hyphen. If it does not, hyphen_ok is set to "0"b (whether or not it was previously "1"b) and control passes to validate the element as a volume name. If the element begins with a hyphen, the value of hyphen_ok is tested. If hyphen_ok = "0"b, the element is assumed to be an attach option and control passes to the attach option validation code. (The first array element can not be an attach option, but <u>must</u> be either -volume, -vol, or a volume name.) If hyphen_ok = "1"b, hyphen_ok is reset to "0"b and control passes to validate the element as a volume name.

The function vname is invoked to validate and normalize the volume name. If the element is not a valid volume name, the procedure returns error_table_$bad_tapeid. If the volume limit is not exceeded, the volume count is incremented and the volume name is placed into the volume name array (tao.volname). If the options array is not exhausted, the next element is tested to determine whether or not it is "-comment" or "-com".

If it is either, no_next is invoked to determine whether or not the next element (the comment text) exists. The length of the comment text is validated and the text is saved in the mount time comment array (tao.comment). Whether or not a comment was processed, control passes to test for a -volume or -vol element, as described above. This algorithm is repeated until either an attach option is encountered or the options array is exhausted.

The attach options are processed by comparing them against a list of valid options and transferring control to the appropriate option processing routines. The actions performed by these routines are best described by the PL/I code itself. Each routine sets a tao structure member to reflect either the appearance of a particular option or its associated value.

Internal Procedures

Entry: no_next

This function is called to determine whether or not the options array contains another element when one is required (For example, -block requires a subsequent element, the block length.) If the current array index plus 1 is greater than the index of the last element, another element does not exist. In this case, error (the diagnostic message) is set equal to the current element (the option requiring the missing element), code is set to error_table_$nodescr, and the procedure returns "1"b. If the next element exists, the array index is incremented and the procedure returns "0"b.

<u>Entry</u>: vname

This function is called to validate and normalize a volume name. If the volume name is longer than six characters, it is invalid. In this case, the function returns a null string and the value "0"b. If the length is exactly six, the volume name is valid and does not require normalization. In this case, the function returns the original volume name and the value "1"b. If the volume name is shorter than six characters, it must be normalized. If the name is entirely numeric, it is normalized by padding on the left with zeros to length six. If it is not entirely numeric, it is normalized by padding on the right with blanks to length six. The function then returns the normalized volume name and the value "1"b.

MODULE: tape_ansi_control_

This module implements the iox_$control function.

<u>Usage</u>

```
     dcl  tape_ansi_control_ entry (ptr, char (*), ptr,
          fixed bin (35));

     call tape_ansi_control_ (iocbP, order, infoP, code);
```

where:

1.	iocbP	is a pointer to the IOCB. (Input)
2.	order	is the control order to be performed. (Input)
3.	infoP	is a pointer to the information structure for a particular order, if required, (Input)
4.	code	is a standard status code. (Output)

The following is a nonexhaustive list of error_table_ codes returned:

not_open	the requested order could not be performed because the I/O switch is not open.
bad_arg	the requested order requires a nonnull information pointer, or the information pointer points to an invalid information structure.

no_operation	the requested order is not implemented.
action_not_performed	the requested order could not be performed. The state of the I/O module (i.e., opening mode, lock value, etc.) did not meet an order-specific criterion.

Internal Logic

The requested order is compared against an array of implemented orders (order_list.name). If no match is found, the procedure returns the error code error_table_$no_operation. Otherwise, order_list.must_be_open is checked to determine whether or not the I/O switch must be open. If it must be open and it is not, the procedure returns error_table_$not_open. The variable order_list.non_null_ptr is tested to determine whether or not the order requires an information structure. If it does and infoP is null the procedure returns error_table_$bad_arg.

If both tests succeed, the cseg pointer is extracted from the IOCB and cseg.invalid is tested to determine whether or not the cseg is valid. If not, the procedure returns error_table_$invalid_cseg. The file lock (cseg.file_lock) is tested to determine whether the file is already in use for other I/O activity. If it is in use, the procedure returns error_table_$file_busy. If it is not in use, a cleanup handler is established and cseg.file_lock is set to "1"b. If invoked, the cleanup handler resets cseg.file_lock to "0"b. The return code is initialized to zero and control transfers to process the particular order requested:

hardware_status

The hardware status string pointed to by infoP is filled with the IOM status bits from the last I/O operation (cseg.hdw_status.bits) and control passes to the exit routine.

status

The status structure (declared by device_status.incl.pl1) pointed to by infoP is filled from the I/O status structured generated by the last I/O operation (cseg.hdw_status) and control passes to the exit routine.

volume_status

The volume status structure (declared by tape_volume_status.incl.pl1) pointed to by infoP is filled with status information describing the "current" volume. If the file set is IBM nonlabeled, the current volume is specified by fd.vlX, the volume currently (or last) in use. If no volume has yet been used (fd.vlX = 0), the first volume of the volume set is the current volume. For ANSI and IBM SL file sets, the current volume is specified by fl.vlX, the volume on which the file section currently (or last) in use resides. If no file section has yet been used (cseg.flP = cseg.fcP), or the file link pointer has been invalidated due to an error (cseg.flP = null), the first volume of the volume set is the current volume. The structure is filled in from the volume link and control passes to the exit routine.

feov

This order forces end of volume on the current volume. If the I/O switch is not open for sequential_output, code is set to error_table_$action_not_performed and control passes to the exit routine. (This order is used only to force a volume switch when writing.) Either tape_ansi_file_cntl_$data_eot or tape_ansi_nl_file_cntl_$data_eot is called to simulate the detection of end-of-tape. The file control procedure performs all necessary volume termination and switching functions. If the returned code is zero, volume switching has been performed successfully and control passes to the exit routine. If the returned code is nonzero, volume switching did not occur, due either to an error or the lack of another volume. In either case, the logical record I/O lock (cseg.lrec.code) is locked to inhibit further I/O, by setting it equal to the returned code. If the returned code is error_table_$no_next_volume, it is set to zero and control passes to the exit routine. Otherwise, an error has occurred and the value of code is passed on to the exit routine.

close_rewind

This order specifies that the current volume is to be rewound when the I/O switch is next closed. The rewind function is performed by the file control procedure. The variable cseg.close_rewind is set to "1"b and control passes to the exit routine.

retention

This order is preserved for historical reasons only. The resource retention variable (cseg.retain) is set to the value of the number pointed to by infoP.

file_status

The file status structure (declared by tape_file_status.incl.pl1) pointed to by infoP is filled in with status information describing the "current" file. If the file set is IBM NL and no file has yet been used, the file status state variable (tape_file_status.state) is set to zero (no information) and control passes to the exit routine. If the file set is ANSI or IBM SL and the file link pointer does not point to a link (cseg.flP = null or cseg.fcP), tape_file_status.state is similarly set to zero and control passes to the exit routine. Even if cseg.flP does point to a link, the file section may not be a part of the attached file. In this case, the above action is also taken.

Once the current file is known, the IOCB open description pointer is checked to determine whether or not the I/O switch is open. If it is not open, tape_file_status.state is set to 1 (not open). If it is open, the state variable is set to either 2 (cseg.lrec.code = 0, logical I/O not locked), or 3 (cseg.lrec.code ^= 0, logical I/O locked.) The remainder of the file status structure is filled in according to whether or not the file set is IBM NL.

retain_none
retain_all

These orders set cseg.retain to 1 (retain neither volumes not devices) or 4 (retain both volumes and devices), respectively. Control then passes to the exit routine. Eventually, the following retain orders should be implemented:

retain_default	cseg.retain = 0
retain_devices	cseg.retain = 2
retain_volumes	cseg.retain = 3

reset_error_lock

If the I/O switch is not open for sequential_input, the return code is set to error_table_$action_not_performed and control passes to the exit routine. If the I/O switch is open for sequential_input, the logical record I/O lock (cseg.lrec.code) is checked to determine if it can be unlocked (cseg.lrec.code = error_table_$tape_error). If it can, the lock is unlocked (set = 0). (If the lock value is already zero, nothing need be done.) Any other lock value causes the return code to be set to error_table_$action_not_performed. Control then passes to the exit routine.

The exit routine sets the file lock to "0"b and returns whatever code has been previously set.

SECTION XII

THE tape_mult_ I/O MODULE

INTRODUCTION

The tape_mult_ I/O module supports I/O to and from Multics standard tapes. (See "Multics Standard Magnetic Tape Format" in Section III of the MPM Peripheral Input/Output Manual, Order No. AX49.

This section will be expanded in a future edition.

SECTION XIII

THE tape_nstd_ I/O MODULE

INTRODUCTION

The tape_nstd_ I/O module supports I/O to and from records on magnetic tape. No logical record or file format is processed or enforced.

This section will be expanded in a future edition.

SECTION XIV

THE rdisk_ I/O MODULE

INTRODUCTION

The rdisk_ I/O module performs explicit I/O on user-attachable disk volumes. These volumes are mounted as "I/O" disks as opposed to storage system disks. Physical operations on the disk are performed via the I/O interfacer ioi_.

This section will be expanded in a future edition.

SECTION XV

THE record_stream_ I/O MODULE

INTRODUCTION

This I/O module associates two I/O switches, causing sequential operations on one switch to generate (or be generated by) corresponding stream operations on the other switch.

PROGRAM MODULES

The record_stream_ I/O module is composed of the following five programs:

record_stream_attach.pl1
: implements attach, detach, open, and close operations. Dispatches to the appropriate module for the opening mode at open and close.

rs_open_str_in.pl1
: implements the get_chars, get_line, and position operations in openings for stream_input.

rs_open_str_out.pl1
: implements the put_chars operation in openings for stream_output.

rs_open_seq_in.pl1
: implements the read_record, read_length, and position operations in openings for sequential_input.

rs_open_seq_out.pl1
: implements the write_record operation in openings for sequential_output.

MODULE record_stream_attach.pl1

Entry: record_stream_attach

This entry point performs the attach operation according to the specified attach options. The attach description is validated and placed in an initialized data block, pointed to by iocb.attach_data_ptr. If the -target option is specified, a uniquely named I/O switch is attached using the remaining options to form the target attach description.

Entry: open_rs

This entry point implements the open operation for all opening modes. The target I/O switch is opened, or if already open, its mode is verified.

Except in the case of openings for sequential output, a uniquely named temporary buffer segment is created and pointed to by iocb.open_data_ptr.

The appropriate module for the given opening mode is called to set up the IOCB entry values for the supported operations, before completing the opening in the common code.

Entry: close_rs

This entry point implements the close operation. In the case of stream_output, the remaining buffer contents (if any) are written out on the target switch. The temporary buffer segment is deleted. If the target switch was initially closed, it is closed again.

Entry: detach_rs

This entry point implements the detach_iocb operation. If the target switch was specified via the -target option, it is detached as well.

<u>Entries</u>: modes_rs, control_rs

These entry points implement the modes and control operations simply by passing the call to the target switch without modification.

MODULE rs_open_str_in.pl1

<u>Entry</u>: get_chars_rs

This entry point implements the get_chars operation. The returned data is copied from the buffer segment, whose initial offset and tail_length are adjusted accordingly. When its contents are exhausted, read_record operations are issued on the target switch into the buffer segment. If the attachment does not specify the -nnl option, a newline character is appended to each record placed in the buffer.

<u>Entry</u>: get_line_rs

This entry point implements the get_line operation similarly to the get_chars operation. The difference is that the length of the returned string is determined via the index of a newline character in the buffer tail.

<u>Entry</u>: position_str_rs

This entry point implements the position operation (except for skipping backwards, which is not supported).

For positioning to either end of the file, the call is simply passed on to the target switch and the buffer contents are discarded.

For skipping forward, the logic is identical to that for the get_line operation, except that no data is copied out of the buffer.

MODULE rs_open_str_out.pl1

Entry: put_chars_rs

This entry point implements the put_chars operation. If the -length (-ln) attach option was specified, fixed length records are written to the target switch as the required number of bytes are made available. The remainder, if any, is appended to the buffer segment, to be written by a subsequent operation.

In the default attachment case, the treatment is similar. Variable-length records are formed from lines with trailing newlines deleted and are written out as they become available. An incomplete line is appended to the buffer and is written on the target switch as part of the next record.

MODULE rs_open_seq_in.pl1

Entry: read_record_rs

This entry point implements the read_record operation. If the buffer segment contains a record, it is returned to the user and the buffer contents are discarded.

If the buffer is empty, a record is obtained directly from the target switch via either a get_chars or get_line operation, depending on the specified attach option.

Entry: read_length_rs

This entry point implements the read_length operation. If the buffer segment contains a record, its length is returned.

Otherwise, a record is read into the buffer from the target switch using either get_line or get_chars, and its length is returned.

Entry: position_seq_rs

This entry point implements the position operation (except for backward skipping).

For positioning to either end of the file, the call is passed directly to the target switch and the buffer segment's contents are discarded.

For skipping forward in the default case, the call is simply passed to the target switch. Otherwise, if the -length (-ln) attach option was specified, records are successively read into the buffer segment until the required number has been skipped or the end of the file is reached. If the buffer segment initially contained a record, the first skip is accomplished by discarding the buffer contents.

MODULE rs_open_seq_out.pl1

Entry: write_record_rs

This entry point implements the write_record operation. No buffer segment is required in this case. A put_chars operation is issued to the target switch with the same arguments as those passed to this entry point. If the -nnl attach option was not specified, a second put_chars operation is issued to the target switch to append a single newline character.

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	LEVEL 68 MULTICS USER RING INPUT/OUTPUT SYSTEM PROGRAM LOGIC MANUAL	ORDER NO.	AN57,Rev. 0
		DATED	MAY 1977

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

31490, 1C581, Printed in U.S.A.