OpenVMS AXP Step 2 Device Driver Reference Manual

Order Number:

August 1993

This manual provides reference material for the *Creating an OpenVMS AXP Step 2 Device Driver from a Step 1 Device Driver* manual. It describes the macro, system routines, and entry points used in converting Step 1 device drivers to Step 2 device drivers.

Note

The information in this manual is preliminary and is subject to change. Change bars indicate sections that have been revised since the previous draft.

Revision/Update Information: This is a new manual.

Software Version: OpenVMS AXP Version 2.0

Digital Equipment Corporation
Maynard, Massachusetts

August 1993

ZK6321

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

Preface ix

1 Driver Macros

CALL_ABORTIO 1–2
CALL_ALLOCBUF, CALL_ALLOCIRP 1–3
CALL_ALLOCEMB 1–4
CALL_ALTQUEPKT 1–5
CALL_ALTREQCOM 1–6
CALL_BROADCAST 1–7
CALL_CANCELIO 1–8
CALL_CARRIAGE 1–9
CALL_CHK*xxx*ACCES 1–10
CALL_CLONE_UCB 1–11
CALL_COPY_UCB 1–12
CALL_CREDIT_UCB 1–13
CALL_CVTLOGPHY 1–14
CALL_CVT_DEVNAM 1–15
CALL_$DELATTNAST 1–16
CALL_DELATTNASTP 1–17
CALL_DELCTRLAST 1–18
CALL_DELCTRLASTP 1–19
CALL_DELETE_UCB 1–20
CALL_DEVICEATTN, CALL_DEVICERR, CALL_DEVICTMO 1–21
CALL_DIAGBUFILL 1–22
CALL_DRVDEALMEM 1–23
CALL_FILSPT 1–24
CALL_FINISHIO, CALL_FINISHIOC, CALL_FINISHIO_NOIOST 1–25
CALL_FLUSHATTNS 1–26
CALL_FLUSHCTRLS 1–27
CALL_GETBYTE 1–28
CALL_INITBUFWIND 1–29
CALL_INITIATE 1–30
CALL_INSERT_IRP 1–31
CALL_IOLOCK 1–32
CALL_IOLOCKR 1–33
CALL_IOLOCKW 1–34
CALL_IORSNWAIT 1–35
CALL_IOUNLOCK 1–36

CALL_LINK_UCB 1–37
CALL_MAPVBLK 1–38
CALL_MNTVER 1–39
CALL_MNTVERSIO 1–40
CALL_MODIFYLOCK, CALL_MODIFYLOCK_ERR 1–41
CALL_MOUNT_VER 1–42
CALL_MOVFRUSER, CALL_MOVFRUSER2 1–43
CALL_MOVTOUSER, CALL_MOVTOUSER2 1–44
CALL_PARSDEVNAM 1–45
CALL_POST, CALL_POST_NOCNT 1–46
CALL_POST_IRP 1–47
CALL_PTETOPFN 1–48
CALL_QIOACPPKT 1–49
CALL_QIODRVPKT 1–50
CALL_QNXTSEG1 1–51
CALL_QXQPPKT 1–52
CALL_READCHK, CALL_READCHKR 1–53
CALL_READLOCK, CALL_READLOCK_ERR 1–54
CALL_RELCHAN 1–55
CALL_RELEASEMB 1–56
CALL_REQCOM 1–57
CALL_SEARCHDEV 1–58
CALL_SEARCHINT 1–59
CALL_SETATTNAST 1–60
CALL_SETCTRLAST 1–61
CALL_SEVER_UCB 1–62
CALL_SIMREQCOM 1–63
CALL_SNDEVMSG 1–64
CALL_THREADCRB 1–65
CALL_UNLOCK 1–66
CALL_WRITECHK, CALL_WRITECHKR 1–67
CALL_WRITELOCK, CALL_WRITELOCK_ERR 1–68
CALL_WRTMAILBOX 1–69
DDTAB 1–70
DPTAB 1–74
DPT_STORE 1–79
DPT_STORE_ISR 1–82
FDT_ACT 1–83
FDT_BUF 1–85
FDT_INI 1–86
FORK 1–87
FORK_ROUTINE 1–89
FORK_WAIT 1–90
IOFORK 1–92
RELCHAN 1–94
REQCHAN 1–95
REQCOM 1–96

REQPCHAN 1–97
WFIKPCH, WFIRLCH 1–98

2 System Routines

ACP_STD$ACCESS 2–2
ACP_STD$ACCESSNET 2–4
ACP_STD$DEACCESS 2–6
ACP_STD$MODIFY 2–8
ACP_STD$MOUNT 2–10
ACP_STD$READBLK 2–12
ACP_STD$WRITEBLK 2–14
COM_STD$DELATTNAST 2–16
COM_STD$DELATTNASTP 2–18
COM_STD$DELCTRLAST 2–20
COM_STD$DELCTRLASTP 2–22
COM_STD$DRVDEALMEM 2–24
COM_STD$FLUSHATTNS 2–26
COM_STD$FLUSHCTRLS 2–28
COM_STD$POST, COM_STD$POST_NOCNT 2–30
COM_STD$SETATTNAST 2–32
COM_STD$SETCTRLAST 2–35
ERL_STD$ALLOCEMB 2–38
ERL_STD$DEVICEATTN, ERL_STD$DEVICERR,
ERL_STD$DEVICTMO 2–40
ERL_STD$RELEASEMB 2–43
EXE_STD$ABORTIO 2–44
EXE_STD$ALLOCBUF, EXE_STD$ALLOCIRP 2–47
EXE_STD$ALTQUEPKT 2–49
EXE_STD$CARRIAGE 2–51
EXE_STD$CHK*xxx*ACCES 2–52
EXE_STD$FINISHIO 2–54
EXE$ILLIOFUNC 2–57
EXE_STD$INSERT_IRP 2–59
EXE_STD$INSIOQ, EXE_STD$INSIOQC 2–61
EXE_STD$IORSNWAIT 2–63
EXE_STD$LCLDSKVALID 2–65
EXE_STD$MNTVERSIO 2–68
EXE_STD$MODIFY 2–70
EXE_STD$MODIFYLOCK 2–74
EXE_STD$MOUNT_VER 2–80
EXE_STD$ONEPARM 2–82
EXE_STD$PRIMITIVE_FORK 2–84
EXE_STD$PRIMITIVE_FORK_WAIT 2–86
EXE_STD$QIOACPPKT 2–88
EXE_STD$QIODRVPKT 2–90
EXE_STD$QUEUE_FORK 2–93

EXE_STD$QXQPPKT 2–94
EXE_STD$READ 2–96
EXE_STD$READCHK 2–100
EXE_STD$READLOCK 2–104
EXE_STD$SENSEMODE 2–110
EXE_STD$SETCHAR, EXE_STD$SETMODE 2–112
EXE_STD$SNDEVMSG 2–115
EXE_STD$WRITE 2–117
EXE_STD$WRITECHK 2–121
EXE_STD$WRITELOCK 2–125
EXE_STD$WRTMAILBOX 2–131
EXE_STD$ZEROPARM 2–133
IOC_STD$ALTREQCOM 2–135
IOC_STD$BROADCAST 2–137
IOC_STD$CANCELIO 2–139
IOC_STD$CLONE_UCB 2–141
IOC_STD$COPY_UCB 2–143
IOC_STD$CREDIT_UCB 2–145
IOC_STD$CVT_DEVNAM 2–146
IOC_STD$CVTLOGPHY 2–148
IOC_STD$DELETE_UCB 2–150
IOC_STD$DIAGBUFILL 2–151
IOC_STD$FILSPT 2–153
IOC_STD$GETBYTE 2–155
IOC_STD$INITBUFWIND 2–157
IOC_STD$INITIATE 2–159
IOC_STD$LINK_UCB 2–162
IOC_STD$MAPVBLK 2–164
IOC_STD$MNTVER 2–166
IOC_STD$MOVFRUSER, IOC_STD$MOVFRUSER2 2–168
IOC_STD$MOVTOUSER, IOC_STD$MOVTOUSER2 2–171
IOC_STD$PARSDEVNAM 2–174
IOC_STD$POST_IRP 2–176
IOC_STD$PTETOPFN 2–177
IOC_STD$QNXTSEG1 2–179
IOC_STD$PRIMITIVE_REQCHANH,
IOC_STD$PRIMITIVE_REQCHANL 2–181
IOC_STD$PRIMITIVE_WFIKPCH,
IOC_STD$PRIMITIVE_WFIRLCH 2–184
IOC_STD$RELCHAN 2–187
IOC_STD$REQCOM 2–189
IOC_STD$SEARCHDEV 2–192
IOC_STD$SEARCHINT 2–194
IOC_STD$SENSEDISK 2–196
IOC_STD$SEVER_UCB 2–198
IOC_STD$SIMREQCOM 2–199
IOC_STD$THREADCRB 2–201

MMG_STD$IOLOCK 2–203
MMG_STD$UNLOCK 2–205
MT_STD$CHECK_ACCESS 2–207
SCH_STD$IOLOCKR 2–209
SCH_STD$IOLOCKW 2–211
SCH_STD$IOUNLOCK 2–213

3 Device Driver Entry Points

Alternate Start-I/O Routine 3–3
Cancel-I/O Routine 3–6
Cancel Selective Routine 3–9
Channel Assign Routine 3–11
Cloned UCB Routine 3–13
Controller Initialization Routine 3–16
Driver Unloading Routine 3–19
FDT Upper-Level Action Routine 3–20
FDT Error-Handling Callback Routine 3–23
Interrupt Service Routine 3–26
Mount Verification Routine 3–29
Register Dumping Routine 3–31
Start-I/O Routine (Traditional) 3–33
Start-I/O Routine (Kernel Process) 3–36
Timeout Handling Code (Traditional) 3–38
Timeout Handling Code (Kernel Process) 3–40
Unit Delivery Routine 3–42
Unit Initialization Routine 3–45

Index

Preface

This manual describes the macros, system routines, and entry points used to convert a Step 1 OpenVMS AXP driver to a Step 2 OpenVMS AXP driver.

Intended Audience

OpenVMS AXP Step 2 Device Driver Reference Manual is intended for software engineers who must prepare a Step 2 device driver to run on OpenVMS AXP Version 2.0.

This manual assumes that its reader is well acquainted with the components of OpenVMS VAX device drivers and Step 1 OpenVMS AXP device drivers. It also relies on a familiarity with the software interfaces within the OpenVMS operating system that support device drivers.

Document Structure

This manual contains the following sections:

- Chapter 1 documents the OpenVMS macros that have been changed or augmented to provide for Step 2 OpenVMS AXP drivers. It also introduces new macros these drivers may use.
- Chapter 2 presents the new call-based OpenVMS system routines that support Step 2 OpenVMS AXP drivers. It also introduces new system routines these drivers may use.
- Chapter 3 provides specific information about how each driver entry point is defined and accessed in a Step 2 OpenVMS AXP driver.

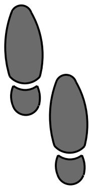
Associated Document

Before reading the *OpenVMS AXP Step 2 Device Driver Reference Manual*, you should understand the material discussed in *Creating a Step 2 Device Driver from a Step 1 Device Driver*.

Conventions

In this manual, every use of OpenVMS VAX means the OpenVMS VAX operating system, every use of OpenVMS AXP means the OpenVMS AXP operating system, and every use of OpenVMS means both the OpenVMS VAX operating system and the OpenVMS AXP operating system.

The following conventions are used to identify information specific to Step 2 OpenVMS AXP drivers.



The two-step icon denotes the beginning of noteworthy information about converting Step 1 OpenVMS AXP device drivers to Step 2 drivers. **This convention is not fully implemented for FT1.**

♦ The diamond symbol denotes the end of a section of Step 2 note material.

The following conventions are used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
$\boxed{\text{Return}}$	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in a OpenVMS file specification, or in the syntax of a substring specification in an assignment statement.)
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in online versions of the manual.
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*), command lines (for example, /PRODUCER=*name*), and command parameters in text.

UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in text are assumed to be decimal, unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Driver Macros

This chapter describes the JSB-replacement macros, FDT completion macros, and other macros used by Step 2 OpenVMS AXP device drivers.

For information about optimizing JSB-replacement macros, see *OpenVMS Step 2 Device Driver Reference Manual*.

CALL_ABORTIO

Completes the servicing of an I/O request without returning status to the I/O status block specified in the request.

Format

CALL_ABORTIO [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

In a Step 2 OpenVMS AXP driver, the CALL_ABORTIO macro simulates the JMP to EXE$ABORTIO in the FDT routine of a Step 1 driver. It initializes the **irp**, **pcb**, **ucb**, and **qio_status** parameters from the contents of R3, R4, R5, and R0, respectively, and calls EXE_STD$ABORTIO. When EXE_STD$ABORTIO returns control to the code generated by a default invocation of CALL_ABORTIO, a RET instruction returns control to the caller of CALL_ABORTIO's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_ALLOCBUF, CALL_ALLOCIRP

Allocates a buffer from nonpaged pool for a buffered-I/O operation.

Format

CALL_ALLOCBUF

CALL_ALLOCIRP

Description

In a Step 2 OpenVMS AXP driver, CALL_ALLOCBUF and CALL_ALLOCIRP simulate a JSB to EXE$ALLOCBUF and EXE$ALLOCIRP, respectively. CALL_ALLOCBUF calls EXE_STD$ALLOCBUF using the current contents of R1 as the **reqsize** argument. Both CALL_ALLOCBUF and CALL_ALLOCIRP return status in R0, the address of the allocated buffer in R2 and its size in R1. If a resource wait occurred, these macros return the address of the PCB in R4.

CALL_ALLOCEMB

Allocates an error message buffer and initializes its header.

Format

CALL_ALLOCEMB

Description

In a Step 2 OpenVMS AXP driver, CALL_ALLOCEMB simulates a JSB to ERL$ALLOCEMB. CALL_ALLOCEMB calls ERL_STD$ALLOCEMB using the current contents of R1 as the **size** argument. It returns status in R0, the address of the allocated EMB in R2 and copies the error log sequence number from EMB$W_DV_ERRSEQ to R1.

CALL_ALTQUEPKT

Delivers an IRP to a driver's alternate start-I/O routine without regard for the status of the device.

Format

CALL_ALTQUEPKT

Description

In a Step 2 OpenVMS AXP driver, the CALL_ALTQUEPKT macro simulates the JSB to EXE$ALTQUEPKT in the FDT routine of a Step 1 driver. CALL_ALTQUEPKT calls EXE_STD$ALTQUEPKT, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

CALL_ALTREQCOM

Completes an I/O request for a device using the disk or tape class drivers.

Format

CALL_ALTREQCOM

Description

In a Step 2 OpenVMS AXP driver, the CALL_ALTREQCOM macro simulates a JSB to IOC$ALTREQCOM in a Step 1 OpenVMS driver. CALL_ALTREQCOM calls IOC_STD$ALTREQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **cdrp** arguments, respectively. When IOC_STD$ALTREQCOM returns, the macro returns the address of the IRP in R3 and the address of the UCB in R4.

CALL_BROADCAST

Broadcasts the specified message to a given terminal.

Format

CALL_BROADCAST [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to IOC_STD$BROADCAST. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_BROADCAST macro simulates a JSB to IOC$BROADCAST in a Step 1 OpenVMS driver. CALL_BROADCAST calls IOC_STD$BROADCAST, using the current contents of R1, R2, and R5 as the **msglen**, **msg_p**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call.

CALL_CANCELIO

Conditionally marks a UCB so that its current I/O request will be canceled.

Format

CALL_CANCELIO   [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to IOC_STD$CANCELIO. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not preserved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_CANCELIO macro simulates a JSB to IOC$CANCELIO in a Step 1 OpenVMS driver. CALL_CANCELIO calls IOC_STD$CANCELIO, using the current contents of R2, R3, R4, and R5 as the **chan**, **irp**, **pcb**, and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_CARRIAGE

Interprets the carriage control specifier in IRP$B_CARCON and converts it to a generic prefix/suffix format.

Format

CALL_CARRIAGE

Description

In a Step 2 OpenVMS AXP driver, the CALL_CARRIAGE macro simulates the JSB to EXE$CARRIAGE in the FDT routine of a Step 1 driver. CALL_CARRIAGE calls EXE_STD$CARRIAGE, using the current contents of R3 as the **irp** arguments.

CALL_CHK*xxx*ACCES

Checks logical (CALL_CHKLOGACCES), physical (CALL_CHKPHYACCES), read (CALL_CHKRDACCES), write (CALL_CHKWRTACCES), execute (CALL_CHKEXEACCES), create (CALL_CHKCREACCES), or delete (CALL_CHKDELACCES) I/O function access, based on the specified protection information.

Format

CALL_CHKCREACCES [save_r1]

CALL_CHKDELACCES [save_r1]

CALL_CHKEXEACCES [save_r1]

CALL_CHKLOGACCES [save_r1]

CALL_CHKPHYACCES [save_r1]

CALL_CHKRDACCES [save_r1]

CALL_CHKWRTACCES [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to EXE_STD$CHKPHYACCES, EXE_STD$CHKLOGACCES, EXE_STD$CHKWRTACCES, EXE_STD$CHKEXEACCES, EXE_STD$CHKCREACCES, EXE_STD$CHKDELACCES or EXE_STD$CHKRDACCES. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_CHKCREACCES, CALL_CHKDELACCES, CALL_CHKEXEACCES, CALL_CHKLOGACCES, CALL_CHKPHYACCES, CALL_CHKWRTACCES, and CALL_CHKRDACCES, macros simulate the JSB to EXE$CHKCREACCES, EXE$CHKDELACCES, EXE$CHKEXEACCES, EXE$CHKPHYACCES, EXE$CHKLOGACCES, EXE$CHKWRTACCES, or EXE$CHKRDACCES in a Step 1 driver. Each macro calls the corresponding access-checking routine, using the current contents of R0, R1, R4, and R5 as the **arb**, **orb**, **pcb**, and **ucb** arguments. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call. All macros return status in R0.

CALL_CLONE_UCB

Copies a template UCB and links it to the appropriate DDB list.

Format

CALL_CLONE_UCB [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. To suppress the warning, specify **interface_warning=NO**.

Description

In a Step 2 OpenVMS AXP driver, CALL_CLONE_UCB simulates a JSB to IOC$CLONE_UCB. It calls IOC_STD$CLONE_UCB using the current contents of R5 as the **tmpl_ucb** argument. CALL_CLONE_UCB returns status in R0 and the address of the newly-created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

CALL_COPY_UCB

Copies and initializes a template UCB and ORB.

Format

CALL_COPY_UCB

Description

In a Step 2 OpenVMS AXP driver, CALL_COPY_UCB simulates a JSB to IOC$COPY_UCB. It calls IOC_STD$COPY_UCB using the current contents of R5 as the **src_ucb** argument. CALL_CLONEUCB returns the address of the newly-created UCB in R2.

CALL_CREDIT_UCB

Credits the UCB charges associated with a given UCB against the process identified by the contents of UCB$L_CPID.

Format

CALL_CREDIT_UCB

Description

In a Step 2 OpenVMS AXP driver, CALL_CREDIT_UCB simulates a JSB to IOC$CREDIT_UCB. It calls IOC_STD$CREDIT_UCB using the current contents of R5 as the **ucb** argument.

CALL_CVTLOGPHY

Conditionally converts a logical block number to a physical disk address and stores the result in the I/O request packet.

Format

CALL_CVTLOGPHY

Description

In a Step 2 OpenVMS AXP driver, the CALL_CVTLOGPHY macro simulates a JSB to IOC$CVTLOGPHY in a Step 1 OpenVMS driver. CALL_CVTLOGPHY calls IOC_STD$CVTLOGPHY, using the current contents of R0, R3, and R5 as the **lbn**, **irp** and **ucb** arguments, respectively.

CALL_CVT_DEVNAM

Converts a device name and unit number to a physical device name string.

Format

CALL_CVT_DEVNAM

Description

In a Step 2 OpenVMS AXP driver, the CALL_CVT_DEVNAM macro simulates a JSB to IOC$CVT_DEVNAM in a Step 1 OpenVMS driver. CALL_CVT_DEVNAM calls IOC_STD$CVT_DEVNAM, using the current contents of R0, R1, R4, and R5 as the **buflen**, **buf**, **form**, and **ucb** arguments, respectively.

The macro returns status in R0 and the length of the conversion string in R1.

CALL_$DELATTNAST

Delivers all attention ASTs linked in the specified list.

Format

CALL_DELATTNAST [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELATTNAST. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_DELATTNAST simulates a JSB to COM$DELATTNAST. CALL_DELATTNAST calls COM_STD$DELATTNAST using the current contents of R4 and R5 as the **listhead** and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DELATTNASTP

Delivers all attention ASTs linked in the specified list for a given process.

Format

CALL_DELATTNASTP [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELATTNASTP. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_DELATTNASTP simulates a JSB to COM$DELATTNASTP. CALL_DELATTNASTP calls COM_STD$DELATTNASTP using the current contents of R4, R5 and R6 as the **listhead**, **ucb**, and **ipid** arguments, respectively. Unless you specify **save_r0r1=NO**, the macro preserves the quadword registers R0 and R1 across the call.

CALL_DELCTRLAST

Delivers all control ASTs, linked in the specified list, that match a given condition.

Format

CALL_DELCTRLAST   [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELCTRLAST. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_DELCTRLAST simulates a JSB to COM$DELCTRLAST. It calls COM_STD$DELCTRLAST using the current contents of R4, R5, and R3 as the **listhead**, **ucb**, and **matchchar** arguments, respectively. When COM$DELCTRLAST returns, it moves the include character into R3. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DELCTRLASTP

Delivers all control ASTs, linked in the specified list, that match a given condition.

Format

CALL_DELCTRLASTP [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELCTRLASTP. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_DELCTRLASTP simulates a JSB to COM$DELCTRLASTP. It calls COM_STD$DELCTRLASTP using the current contents of R4, R5, R6, and R3 as the **listhead**, **ucb**, **ipid**, and **matchchar** arguments, respectively. When COM$DELCTRLASTP returns, it moves the include character into R3. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DELETE_UCB

Deletes the specified UCB if its reference count is zero and UCB$V_DELETEUCB is set in UCB$L_STS.

Format

CALL_DELETE_UCB

Description

In a Step 2 OpenVMS AXP driver, CALL_DELETE_UCB simulates a JSB to IOC$DELETE_UCB. It calls IOC_STD$DELETE_UCB using the current contents of R5 as the **ucb** argument.

CALL_DEVICEATTN, CALL_DEVICERR, CALL_DEVICTMO

Allocate an error message buffer and record in it information concerning the error.

Format

CALL_DEVICEATTN [save_r0r1]

CALL_DEVICERR [save_r0r1]

CALL_DEVICTMO [save_r0r1]

Parameters

save_r0r1
Indicates that the macros must preserve the contents of R0 and R1 across the call to ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, or ERL_STD$DEVICTMO. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_DEVICEATTN, CALL_DEVICERR, and CALL_DEVICTMO macros simulate JSBs to ERL$DEVICEATTN, ERL$DEVICERR, and ERL$DEVICTMO, respectively. Each macro calls the corresponding routine using the current contents of R4 and R5 as the **driver_param** and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DIAGBUFILL

Fills a diagnostic buffer if the original $QIO request specified such a buffer.

Format

CALL_DIAGBUFILL

Description

In a Step 2 OpenVMS AXP driver, the CALL_DIAGBUFILL macro simulates a JSB to IOC$DIAGBUFILL in a Step 1 OpenVMS driver. CALL_DIAGBUFILL calls IOC_STD$DIAGBUFILL, using the current contents of R4 and R5 as the **driver_param** and **ucb** arguments, respectively.

CALL_DRVDEALMEM

Deallocates system dynamic memory.

Format

CALL_DRVDEALMEM [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DRVDEALMEM. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_DRVDEALMEM simulates a JSB to COM$DRVDEALMEM. It calls COM_STD$DRVDEALMEM using the current contents of R0 as the **ptr** argument. Unless you specify **save_r0r1=NO**, the macro preserves the quadword registers R0 and R1 across the call.

CALL_FILSPT

Fills a system page-table entry (PTE) with the transfer PTE of a buffer that is locked in memory so that the system PTE may be directly addressed.

Format

CALL_FILSPT

Description

In a Step 2 OpenVMS AXP driver, CALL_FILSPT simulates a JSB to IOC$FILSPT. CALL_FILSPT calls IOC_STD$FILSPT, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

CALL_FINISHIO, CALL_FINISHIOC, CALL_FINISHIO_NOIOST

Complete the servicing of an I/O request and return status to the I/O status block specified in the original call to the $QIO system service.

Format

CALL_FINISHIO [do_ret=YES]

CALL_FINISHIOC [do_ret=YES]

CALL_FINISHIO_NOIOST [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

In a Step 2 OpenVMS AXP driver, the CALL_FINISHIO macro simulates the JMP to EXE$FINISHIO in the FDT routine of a Step 1 driver. The CALL_FINISHIOC macro simulates the JMP to EXE$FINISHIOC, and CALL_FINISHIO_NOIOST the JMP to EXE$FINISHIO_NOIOST. CALL_FINISHIO moves the current contents of R0 and R1 into IRP$L_IOST1 and IRP$L_IOST2, respectively; CALL_FINISHIOC initializes IRP$L_IOST1 from R0 and clears IRP$L_IOST2; and CALL_FINISHIO_NOIOST fills in neither IRP field. The macros initialize the **irp** and **ucb** parameters from the contents of R3 and R5, respectively before calling EXE_STD$FINISHIO. When EXE_STD$FINISHIO returns control to the code generated by a default invocation of CALL_FINISHIO, CALL_FINISHIOC, or CALL_FINISHIO_NOIOST, a RET instruction returns control to the caller of the macro's invoker.

Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_FLUSHATTNS

Removes specified ASTs from an attention AST list.

Format

CALL_FLUSHATTNS

Description

In a Step 2 OpenVMS AXP driver, CALL_FLUSHATTNS simulates a JSB to COM$FLUSHATTNS. It calls COM_STD$FLUSHATTNS using the current contents of R4, R5, R6, and R7 as the **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

CALL_FLUSHCTRLS

Removes specified ASTs from a control AST list.

Format

CALL_FLUSHCTRLS

Description

In a Step 2 OpenVMS AXP driver, CALL_FLUSHCTRLS simulates a JSB to COM$FLUSHCTRLS. It calls COM_STD$FLUSHCTRLS using the current contents of R2, R4, R5, R6, and R7 as the **mask**, **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

CALL_GETBYTE

Fetches a single byte of data from a user buffer.

Format

CALL_GETBYTE

Description

In a Step 2 OpenVMS AXP driver, CALL_GETBYTE simulates a JSB to IOC$GETBYTE. $GETBYTE calls IOC_STD$GETBYTE, passing the current contents of R0 and R5 as the **sva** and **ucb** arguments, respectively. It returns in R0 the byte of data (not zero-extended) returned from the user buffer. It returns in R1 the updated system virtual address. (Note that this differs from the behavior of IOC$GETBYTE, which returns the byte of data in R1 and the updated system virtual address in R0.)

CALL_INITBUFWIND

Initializes a single-page window into a user buffer.

Format

CALL_INITBUFWIND

Description

In a Step 2 OpenVMS AXP driver, CALL_INITBUFWIND simulates a JSB to IOC$INITBUFWIND. CALL_INITBUFWIND calls IOC_STD$INITBUFWIND, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

CALL_INITIATE

Initiates the processing of the next I/O request for a device unit.

Format

CALL_INITIATE

Description

In a Step 2 OpenVMS AXP driver, the CALL_INITIATE macro simulates a JSB to IOC$INITIATE in a Step 1 OpenVMS driver. CALL_INITIATE calls IOC_STD$INITIATE, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

CALL_INSERT_IRP

Inserts an I/O request packet (IRP) into the specified queue of IRPs according to the base priority of the process that issued the I/O request.

Format

CACLL_INSERT_IRP

Description

In a Step 2 OpenVMS AXP driver, the CALL_INSERT_IRP macro simulates a JSB to EXE$INSERT_IRP in a Step 1 OpenVMS AXP driver. CALL_INSERT_IRP calls EXE_STD$INSERT_IRP, using the current contents of R2 and R3 as the **irp_lh** and **irp** arguments, respectively. It returns status in R0.

CALL_IOLOCK

Locks process pages in memory.

Format

CALL_IOLOCK

Description

In a Step 2 OpenVMS AXP driver, CALL_IOLOCK simulates a JSB to MMG$IOLOCK. It calls MMG_STD$IOLOCK using the current contents of R0, R1, R2, and R4 as the **buf**, **bufsize**, **is_read**, and **pcb** arguments, respectively.

CALL_IOLOCK returns status in R0. If R0 contains SS$_NORMAL, R1 contains the system virtual address of the first page-table entry. If R0 contains zero, R1 contains the address of a page to be faulted into memory. R0 can also contain a system-level status.

CALL_IOLOCKR

Locks the I/O database mutex on behalf of its caller for read access.

Format

CALL_IOLOCKR save_r1

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to SCH_STD$IOLOCKR. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_IOLOCKR simulates a JSB to SCH$IOLOCKR. It calls SCH_STD$IOLOCKR using the current contents of R4 as the **pcb** argument.

CALL_IOLOCKR returns the address of the I/O database mutex in R0. Unless you specify **save_r1=NO**, the macro preserves R1 across the call.

CALL_IOLOCKW

Locks the I/O database mutex on behalf of its caller for write access.

Format

CALL_IOLOCKW save_r1

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to SCH_STD$IOLOCKW. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_IOLOCKW simulates a JSB to SCH$IOLOCKW. It calls SCH_STD$IOLOCKW using the current contents of R4 as the **pcb** argument.

CALL_IOLOCKW returns the address of the I/O database mutex in R0. Unless you specify **save_r1=NO**, the macro preserves R1 across the call.

CALL_IORSNWAIT

Places a process in a resource wait state if it has enabled resource waits.

Format

CALL_IORSNWAIT [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

In a Step 2 OpenVMS AXP driver, the CALL_IORSNWAIT macro simulates the JMP to EXE$IORSNWAIT in the FDT routine of a Step 1 driver. It calls EXE_STD$IORSNWAIT using the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **qio_status**, and **rsn** arguments, respectively. When EXE_STD$IORSNWAIT returns control to the code generated by a default invocation of $IORSNWAIT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_IOUNLOCK

Releases ownership of the I/O database mutex and, if the mutex has thus become available to waiting processes, reactivates the next eligible process.

Format

CALL_IOUNLOCK

Description

In a Step 2 OpenVMS AXP driver, CALL_IOUNLOCK simulates a JSB to SCH$IOUNLOCK. It calls SCH_STD$IOUNLOCK using the current contents of R4 as the **pcb** argument.

CALL_LINK_UCB

Searches the UCB list attached to the device data block identified by the specified UCB and links the specified UCB into the list in ascending unit number order.

Format

CALL_LINK_UCB [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

Description

In a Step 2 OpenVMS AXP driver, CALL_LINK_UCB simulates a JSB to IOC$LINK_UCB. It calls IOC_STD$LINK_UCB using the current contents of R5 as the **ucb** argument. CALL_LINK_UCB returns the status in R0 and address of the newly-created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

CALL_MAPVBLK

Maps a virtual block to a logical block using a mapping window.

Format

CALL_MAPVBLK

Description

In a Step 2 OpenVMS AXP driver, the CALL_MAPVBLK macro simulates a JSB to IOC$MAPVBLK in a Step 1 OpenVMS driver. CALL_MAPVBLK calls IOC_STD$MAPVBLK, using the current contents of R0, R1, R2, R3, and R5 as the **vbn**, **numbytes**, **wcb**, **irp** and **ucb** arguments, respectively. It returns status in R0, the address of the logical block number of the first block mapped in R1, the number of unmapped bytes in R2, and the address of the updated UCB in R3. If the low bit of the status value in R0 is clear, signifying failure status, only the value in R2 is valid.

CALL_MNTVER

Assists a driver with mount verification.

Format

CALL_MNTVER

Description

In a Step 2 OpenVMS AXP driver, the CALL_MNTVER macro simulates a JSB to IOC$MNTVER in a Step 1 OpenVMS driver. CALL_MNTVER calls IOC_STD$MNTVER, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

CALL_MNTVERSIO

Processes I/O functions that affect the online count and local valid status of a disk.

Format

CALL_MNTVERSIO

Description

In a Step 2 OpenVMS AXP driver, the CALL_MNTVERSIO macro simulates a JSB to EXE$MNTVERSIO in a Step 1 OpenVMS AXP driver. CALL_MNTVERSIO calls EXE_STD$MNTVERSIO, using the current contents of R0, R3, and R5 as the **rout**, **irp**, and **ucb** arguments, respectively.

CALL_MODIFYLOCK, CALL_MODIFYLOCK_ERR

Validate and prepare a user buffer for a direct-I/O, DMA read/write operation.

Format

CALL_MODIFYLOCK

CALL_MODIFYLOCK_ERR [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

Description

In a Step 2 OpenVMS AXP driver, CALL_MODIFYLOCK simulates a JSB to EXE$MODIFYLOCK and CALL_MODIFYLOCK_ERR simulates a JSB to EXE$MODIFYLOCK_ERR. CALL_MODIFYLOCK calls EXE_STD$MODIFYLOCK, specifying 0 as the **err_rout** argument; CALL_MODIFYLOCK_ERR also calls EXE_STD$MODIFYLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$MODIFYLOCK or EXE_STD$MODIFYLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and writes a 5 into R2 to indicate a modify operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 5 to R2 to indicate a modify operation and and returns to FDT dispatching code in the $QIO system service.

CALL_MOUNT_VER

During I/O postprocessing, determines whether mount verification should be initiated on a given disk or tape device on behalf of the I/O request being completed.

Format

CALL_MOUNT_VER [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to EXE_STD$MOUNT_VER. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_MOUNT_VER macro simulates a JSB to EXE$MOUNT_VER in a Step 1 OpenVMS driver. CALL_MOUNT_VER calls EXE_STD$MOUNT_VER, using the current contents of R0, R1, R3, and R5 as the **iost1**, **iost2**, **irp**, and **ucb** arguments, respectively. When EXE_STD$MOUNT_VER returns, code generated by this macro copies return status from R0 to R2. Unless you specify **save_r0r1=NO**, the macro preserves the quadword registers R0 and R1 across the call.

CALL_MOVFRUSER, CALL_MOVFRUSER2

Move data from a user buffer to a device.

Format

CALL_MOVFRUSER

CALL_MOVFRUSER2

Description

In a Step 2 OpenVMS AXP driver, CALL_MOVFRUSER and CALL_MOVFRUSER2 simulate a JSB to IOC$MOVFRUSER and IOC$MOVFRUSER2 respectively. CALL_MOVFRUSER calls IOC_STD$MOVFRUSER, and CALL_MOVFRUSER2 calls IOC_STD$MOVFRUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. CALL_MOVFRUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

CALL_MOVTOUSER, CALL_MOVTOUSER2

Move data from an internal buffer to a user buffer.

Format

CALL_MOVTOUSER

CALL_MOVTOUSER2

Description

In a Step 2 OpenVMS AXP driver, CALL_MOVTOUSER and CALL_MOVTOUSER2 simulate a JSB to IOC$MOVTOUSER and IOC$MOVTOUSER2 respectively. CALL_MOVTOUSER calls IOC_STD$MOVTOUSER, and CALL_MOVTOUSER2 calls IOC_STD$MOVTOUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. CALL_MOVTOUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

CALL_PARSDEVNAM

Parses a device name string, checking its syntax and extracting the node name, allocation class number, and unit number.

Format

CALL_PARSDEVNAM

Description

In a Step 2 OpenVMS AXP driver, the CALL_PARSDEVNAM macro simulates a JSB to IOC$PARSDEVNAM in a Step 1 OpenVMS driver. CALL_PARSDEVNAM calls IOC_STD$PARSDEVNAM, using the current contents of R8, R9, and R10 as the **devnamsiz**, **devnam**, and **flags** arguments, respectively. When IOC_STD$PARSDEVNAM returns, the macro returns status in R0; the unit number in R2; the length of the SCS node name at the beginning of the name string, allocation class number, or device type code in R3; the size of the name string in R8, the address of the name string in R9, and the flags in R10.

CALL_POST, CALL_POST_NOCNT

Initiate device-independent postprocessing of an I/O request independent of the status of the device unit.

Format

CALL_POST [save_r1]

CALL_POST_NOCNT [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to COM_STD$POST or COM_STD$POST_NOCNT. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_POST simulates a JSB to COM$POST. It calls COM_STD$POST using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. CALL_POST_NOCNT simulates a JSB to COM$POST_NOCNT. It calls COM_STD$POST_NOCNT using the current contents of R3 as the **irp** argument. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call.

CALL_POST_IRP

Inserts an I/O request packet in a CPU-specific I/O postprocessing queue.

Format

CALL_POST_IRP

Description

In a Step 2 OpenVMS AXP driver, CALL_POST_IRP simulates a JSB to IOC$POST_IRP. It calls IOC_STD$POST_IRP using the current contents of R3 as the **irp** argument.

CALL_PTETOPFN

Returns a page frame number (PFN) from a page-table entry (PTE) that has already been determined to be invalid.

Format

CALL_PTETOPFN

Description

In a Step 2 OpenVMS AXP driver, CALL_PTETOPFN simulates a JSB to IOC$PTETOPFN. CALL_PTETOPFN extracts the quadword page-table entry from R3 and passes a pointer to it as the **pte** argument to IOC_STD$PTETOPFN. It returns the page frame number in R0.

CALL_QIOACPPKT

Delivers an IRP to the appropriate ACP or XQP.

Format

CALL_QIOACPPKT [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

In a Step 2 OpenVMS AXP driver, the CALL_QIOACPPKT macro simulates the JMP to EXE$QIOACPPKT in the FDT routine of a Step 1 driver. It calls EXE_STD$QIOACPPKT using the current contents of R3, R4, and R5 as the **irp**, **pcb**, and **ucb** arguments, respectively. When EXE_STD$QIOACPPKT returns control to the code generated by a default invocation of $QIOACPPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_QIODRVPKT

Delivers an IRP to the driver's start-I/O routine or pending-I/O queue.

Format

CALL_QIODRVPKT [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

In a Step 2 OpenVMS AXP driver, the CALL_QIODRVPKT macro simulates the JMP to EXE$QIODRVPKT in the FDT routine of a Step 1 driver. CALL_QIODRVPKT clears IRP$PS_FDT_CONTEXT and calls EXE_STD$INSIOQ, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. When EXE_STD$INSIOQ returns control to the code generated by a default invocation of CALL_QIODRVPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0.

CALL_QNXTSEG1

Queues the next segment of a virtual I/O request that did not map to a single contiguous I/O request.

Format

CALL_QNXTSEG1

Description

In a Step 2 OpenVMS AXP driver, CALL_QNXTSEG1 simulates a JSB to IOC$QNXTSEG1. It calls IOC_STD$QNXTSEG1 using the current contents of R0, R1, R2, R3, R4, and R5 as the **vbn**, **bcnt**, **wcb**, **irp**, **pcb**, and **ucb** arguments. It returns the address of the updated UCB in R5.

CALL_QXQPPKT

Inserts an IRP on the tail of the XQP work queue and initiates its processing if it is the only request on the queue.

Format

CALL_QXQPPKT

Description

In a Step 2 OpenVMS AXP driver, the CALL_QXQPPKT macro simulates the JMP to EXE$QXQPPKT in the FDT routine of a Step 1 driver. It calls EXE_STD$QXQPPKT using the current contents of R4 and R5 as the **pcb** and **acb** arguments, respectively. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_READCHK, CALL_READCHKR

Verifies that a process has write access to the pages in the buffer specified in a $QIO request.

Format

CALL_READCHK

Description

In a Step 2 OpenVMS AXP driver, CALL_READCHK simulates a JSB to EXE$READCHK and CALL_READCHKR simulates a JSB to EXE$READCHKR. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$READCHK returns, CALL_READCHK and CALL_READCHKR move 1 into R2 to indicate a read operation and examines the return status:

- If success status (SS$_NORMAL) is returned, CALL_READCHK and CALL_READCHKR copy the contents of IRP$L_BCNT into R1. CALL_READCHK writes the starting address of the I/O buffer in R0; CALL_READCHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, CALL_READCHK returns to FDT dispatching code in the $QIO system service. CALL_READCHKR does not return control to $QIO.

CALL_READLOCK, CALL_READLOCK_ERR

Validate and prepare a user buffer for a direct-I/O, DMA write operation.

Format

CALL_READLOCK

CALL_READLOCK_ERR [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

Description

In a Step 2 OpenVMS AXP driver, the CALL_READLOCK simulates a JSB to EXE$READLOCK and CALL_READLOCK_ERR simulates a JSB to EXE$READLOCK_ERR. CALL_READLOCK calls EXE_STD$READLOCK, specifying 0 as the **err_rout** argument; CALL_READLOCK_ERR also calls EXE_STD$READLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$READLOCK or EXE_STD$READLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro copies the contents of IRP$L_SVAPTE into R1 and writes a 1 to R2 to indicate a read operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 1 to R2 to indicate a read operation and returns to FDT dispatching code in the $QIO system service.

CALL_RELCHAN

Releases device ownership of all controller data channels.

Format

CALL_RELCHAN

Description

In a Step 2 OpenVMS AXP driver, CALL_RELCHAN simulates a JSB to IOC$RELCHAN. It calls IOC_STD$RELCHAN using the current contents of R5 as the **ucb** argument.

CALL_RELEASEMB

Releases an error message buffer to the error logging process.

Format

CALL_RELEASEMB

Description

In a Step 2 OpenVMS AXP driver, CALL_RELEASEMB simulates a JSB to ERL$RELEASEMB. It calls ERL_STD$RELEASEMB using the current contents of R2 as the **embdv** argument.

CALL_REQCOM

Completes an I/O operation on a device unit, requests I/O postprocessing of the current request, and starts the next I/O request waiting for the device.

Format

CALL_REQCOM

Description

In a Step 2 OpenVMS AXP driver, the CALL_REQCOM macro simulates a JSB to IOC$REQCOM in a Step 1 OpenVMS driver. CALL_REQCOM calls IOC_STD$REQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **ucb** arguments, respectively.

CALL_SEARCHDEV

Searches the I/O database for a specific physical device.

Format

CALL_SEARCHDEV

Description

In a Step 2 OpenVMS AXP driver, the CALL_SEARCHDEV macro simulates a JSB to IOC$SEARCHDEV in a Step 1 OpenVMS driver. $SEARCHDEV calls IOC_STD$SEARCHDEV, using the current contents of R1 as the **descr_p** argument. When IOC_STD$SEARCHDEV returns, the macro returns returns status in R0, the UCB address in R1, the DDB address in R2, and the SB address in R3.

CALL_SEARCHINT

Searches the I/O database for the specified device, using specified search rules.

Format

CALL_SEARCHINT

Description

In a Step 2 OpenVMS AXP driver, the CALL_SEARCHINT macro simulates a JSB to IOC$SEARCHINT in a Step 1 OpenVMS driver. CALL_SEARCHINT calls IOC_STD$SEARCHINT, using the current contents of R2, R3, R8, R9 and R10 as the **unit**, **scslen**, **devnamlen**, **devnam**, and **flags** arguments, respectively. When IOC_STD$SEARCHINT returns, the macro returns status in R0, the UCB address in R5, the DDB address in R6, and the SB address in R7.

CALL_SETATTNAST

Enables or disables attention ASTs.

Format

CALL_SETATTNAST

Description

In a Step 2 OpenVMS AXP driver, CALL_SETATTNAST simulates a JSB to COM$SETATTNAST. It calls COM_STD$SETATTNAST using the current contents of R3, R4, R5, R6, and R7, as the **irp**, **pcb**, **ucb**, **ccb**, and **acb_lh** arguments, respectively. It returns status in R0 and in the FDT_CONTEXT structure.

CALL_SETCTRLAST

Enables or disables control ASTs.

Format

CALL_SETCTRLAST

Description

In a Step 2 OpenVMS AXP driver, CALL_SETCTRLAST simulates a JSB to COM$SETCTRLAST. It calls COM_STD$SETCTRLAST using the current contents of R3, R4, R5, R7, and R2, as the **irp**, **pcb**, **ucb**, **acb_lh**, and **mask** arguments, respectively. It returns the TAST block in R2. It returns status in R0 and in the FDT_CONTEXT structure.

CALL_SEVER_UCB

Removes the specified UCB from the UCB list of the device data block identified within the specified UCB.

Format

CALL_SEVER_UCB

Description

In a Step 2 OpenVMS AXP driver, CALL_SEVER_UCB simulates a JSB to IOC$SEVER_UCB. It calls IOC_STD$SEVER_UCB using the current contents of R5 as the **ucb** argument.

CALL_SIMREQCOM

Completes an I/O operation by setting an event flag, modifying an I/O status block (IOSB), setting an event flag, or queuing an AST to the process requesting the I/O. The caller of this routine is responsible for checking quotas and updating the I/O count.

Format

CALL_SIMREQCOM

Description

In a Step 2 OpenVMS AXP driver, the CALL_SIMREQCOM macro simulates a JSB to IOC$SIMREQCOM in a Step 1 OpenVMS driver. CALL_SIMREQCOM calls IOC_STD$SIMREQCOM, using the current contents of R1, R2, R3, R4, R5, and R6 as the **iosb**, **pri**, **efn**, **iost**, **acb**, and **acmode** arguments, respectively.

CALL_SNDEVMSG

Builds and sends a device-specific message to the mailbox of a system process, such as the job controller or OPCOM.

Format

CALL_SNDEVMSG [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_SNDEVMSG macro simulates a JSB to EXE$SNDEVMSG in a Step 1 OpenVMS driver. CALL_SNDEVMSG calls EXE_STD$SNDEVMSG, using the current contents of R3, R4, and R5 as the **mb_ucb**, **msgtyp**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1=NO**, the macro preserves the R1 across the call.

CALL_THREADCRB

Threads a controller request block (CRB) onto the due-time chain headed by IOC$GL_CRBTMOUT.

Format

CALL_THREADCRB [save_r0]

Parameters

save_r0
Indicates that the macro must preserve the contents of R0 across the call to IOC_STD$THREADCRB. If **save_r0** is blank or **save_r0=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r0=NO**, R0 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, CALL_THREADCRB simulates a JSB to IOC$THREADCRB. It calls IOC_STD$THREADCRB using the current contents of R3 as the **crb** argument. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call.

CALL_UNLOCK

Unlocks process pages previously locked for a direct-I/O operation.

Format

CALL_UNLOCK

Description

In a Step 2 OpenVMS AXP driver, CALL_UNLOCK simulates a JSB to MMG$UNLOCK. It calls MMG_STD$UNLOCK using the current contents of R1 and R3 as the **npages** and **svapte** arguments, respectively.

CALL_WRITECHK, CALL_WRITECHKR

Verify that a process has read access to the pages in the buffer specified in a $QIO request.

Format

CALL_WRITECHK

CALL_WRITECHKR

Description

In a Step 2 OpenVMS AXP driver, CALL_WRITECHK simulates a JSB to EXE$WRITECHK and CALL_READCHKR simulates a JSB to EXE$READCHKR. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$WRITECHK returns, CALL_WRITECHK and CALL_WRITECHKR clear R2 to indicate a write operation and examines the return status:

- If success status (SS$_NORMAL) is returned, CALL_WRITECHK and CALL_WRITECHKR copy the contents of IRP$L_BCNT into R1. CALL_WRITECHK writes the starting address of the I/O buffer in R0; CALL_WRITECHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, CALL_WRITECHK returns to FDT dispatching code in the $QIO system service. CALL_WRITECHKR does not return control to $QIO.

CALL_WRITELOCK, CALL_WRITELOCK_ERR

Validate and prepare a user buffer for a direct-I/O, DMA read operation.

Format

CALL_WRITELOCK

CALL_WRITELOCK_ERR [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

Description

In a Step 2 OpenVMS AXP driver, the CALL_WRITELOCK simulates a JSB to EXE$WRITELOCK and CALL_WRITELOCK_ERR simulates a JSB to EXE$WRITELOCK_ERR. CALL_WRITELOCK calls EXE_STD$WRITELOCK, specifying 0 as the **err_rout** argument; CALL_WRITELOCK_ERR also calls EXE_STD$WRITELOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$WRITELOCK or EXE_STD$WRITELOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and clears R2 to indicate a write operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro clears R2 to indicate a write operation and returns to FDT dispatching code in the $QIO system service.

CALL_WRTMAILBOX

Sends a message to a mailbox.

Format

CALL_WRTMAILBOX [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

Description

In a Step 2 OpenVMS AXP driver, the CALL_WRTMAILBOX macro simulates a JSB to EXE$WRTMAILBOX in a Step 1 OpenVMS driver. CALL_WRTMAILBOX calls EXE_STD$WRTMAILBOX, using the current contents of R5, R3, and R4 as the **mb_ucb**, **msgsiz**, and **msg** arguments, respectively. It returns status in R0. Unless you specify **save_r1=NO**, the macro preserves the R1 across the call.

DDTAB

Generates a driver dispatch table (DDT) labeled *devnam*$DDT.

Format

```
DDTAB  devnam ,[start=IOC$RETURN_SUCCESS]
       ,[ctrlinit=IOC$RETURN_SUCCESS] ,functb
       [,cancel=IOC$RETURN_SUCCESS] [,regdmp=IOC$RETURN_SUCCESS]
       [,diagbf=0] [,erlgbf=0] [,unitinit=IOC$RETURN_SUCCESS]
       [,altstart=IOC$RETURN_SUCCESS] [,mntver=IOC_STD$MNTVER]
       [,cloneducb=IOC$RETURN_SUCCESS]
       [,mntv_sssc=IOC$RETURN_SUCCESS]
       [,mntv_for=IOC$RETURN_SUCCESS]
       [,mntv_sqd=IOC$RETURN_SUCCESS]
       [,channel_assign=IOC$RETURN_SUCCESS]
       [,cancel_selective=IOC$RETURN_SUCCESS] [,kp_stack_size=0]
       [,kp_reg_mask=0] [,kp_startio=IOC$RETURN_SUCCESS] [,aux_storage=0]
       [,aux_routine=IOC$RETURN_SUCCESS] [,step]
```

Parameters

devnam
Generic name of the device.

[start=IOC$RETURN_SUCCESS]
Address of the driver's start-I/O routine. For drivers that use the kernel process services, this is the address of the kernel process start-I/O routine (EXE_STD$KP_STARTIO).

[ctrlinit=IOC$RETURN_SUCCESS]
Address of the controller initialization routine.

functb
Address of the driver's function decision table (FDT).

[cancel=IOC$RETURN_SUCCESS]
Address of the cancel-I/O routine. Many drivers specify the address of the system cancel-I/O routine (IOC_STD$CANCELIO) in this argument.

[regdmp=IOC$RETURN_SUCCESS]
Address of the routine that dumps the device registers to an error message buffer or to a diagnostic buffer.

[diagbf=0]
Length in bytes of the diagnostic buffer.

[erlgbf=0]
Length in bytes of the error message buffer.

[unitinit=IOC$RETURN_SUCCESS]
Address of the unit initialization routine.

[altstart=IOC$RETURN_SUCCESS]
Address of the alternate start-I/O routine.

[mntver=IOC_STD$MNTVER]
Address of the system-provided routine that is called at the beginning and end of a mount verification operation. The default, IOC_STD$MNTVER, is suitable for all single-stream disk drives. Use of this argument to call any other routine is reserved to Digital.

[cloneducb=IOC$RETURN_SUCCESS]
Address of the routine called when a UCB is cloned by the $ASSIGN system service.

[mntv_sssc=IOC$RETURN_SUCCESS]
Address of the routine called when the system performs mount verification for a shadow set state change. This argument is reserved to Digital.

[mntv_for=IOC$RETURN_SUCCESS]
Address of the routine called when the system performs mount verification for a foreign device. This argument is reserved to Digital.

[mntv_sqd=IOC$RETURN_SUCCESS]
Address of the routine called when the system performs mount verification for a sequential device. This argument is reserved to Digital.

[channel_assign=IOC$RETURN_SUCCESS]
Address of the routine, called by SYS$ASSIGN, to complete channel assignment in a device-specific manner. This argument is reserved to Digital. (Note that channel-assignment routines are not yet implemented on OpenVMS AXP systems.)

[cancel_selective=IOC$RETURN_SUCCESS]
Address of the routine that cancels a list of I/O requests from the specified channel, including both waiting and active requests. This argument is reserved to Digital. (Note that cancel selective routines are not yet implemented on OpenVMS AXP systems.)

[kp_stack_size=0]
Size in bytes of the kernel process stack. EXE_STD$KP_STARTIO uses this value, or KPB$K_MIN_IO_STACK (currently 8KB), whichever is larger, to determine the size of the stack created for the driver's start I/O kernel process thread.

[kp_reg_mask=0]
Kernel process register save mask.

This mask represents those registers used by a kernel process that must be preserved across kernel process context switches. Note that R12 through R15, R26, R27, and R29 (KPREG$K_MIN_REG_MASK) are always preserved across kernel process context switches, and that EXE$KP_STARTIO additionally includes R2 through R5 in this register set (KPREG$K_MIN_IO_REG_MASK). R0, R1, R16 through R25, R27, R28, R30, and R31 (KPREG$K_ERR_REG_MASK) are never preserved and are illegal in a register save mask.

[kp_startio=IOC$RETURN_SUCCESS]
Address of the start-I/O routine of a driver that employs the kernel process services. Such a driver typically specifies the system routine EXE_STD$KP_STARTIO in the **start** argument to the DDTAB macro. EXE_STD$KP_STARTIO

calls the start-I/O routine specified in this argument after setting up the kernel process environment.

[aux_storage=0]
Address of auxiliary storage area. This argument is reserved to Digital. (Note that auxiliary storage areas are not yet implemented on OpenVMS AXP systems.)

[aux_routine=IOC$RETURN_SUCCESS]
Address of an auxiliary routine in the OpenVMS VAX mailbox driver that is called by SYS$ASSIGN. This argument is reserved to Digital. (Note that auxiliary routines are not yet implemented on OpenVMS AXP systems.)

[step]
OpenVMS AXP driver step number. You may indicate that a given driver conforms to the coding practices for an Step 1 OpenVMS AXP device driver by supplying **step=2** in the DDTAB macro invocation. If you previously specified the **step** argument to the DPTAB macro, you need not repeat it here.

If you supply the **step** argument, but specify a value other than 1 or 2, the DPTAB macro generates the following message:

```
%MACRO-E-GENERR, Generated ERROR: DDTAB must declare driver STEP=1 or STEP=2
```

Step 2 drivers typically supply a value for the **step** argument of the DPTAB macro. If the step values given the DPTAB and DDTAB macros conflict, the DDTAB macro generates the error:

```
%MACRO-E-GENERR, Generated ERROR: DDTAB STEP=x conflicts with prior declaration.
```

Description

The DDTAB macro creates a driver dispatch table (DDT), using the DRIVER_ DATA macro to place it within the driver's data program section ($$$110_DATA). The macro assigns the table a label in the form of **devnam**$DDT.

The DDTAB macro writes the address of the universal executive routine vector IOC$RETURN_SUCCESS into routine address fields of the DDT that are not supplied in the macro invocation (with the exception of the **mntver** argument). IOC$RETURN_SUCCESS places success status in R0 and issues an RSB instruction.

Example

```
DDTAB      -                        ;DDT-creation macro
DEVNAM=XX, -                        ;Name of device
START=XX_START,-                    ;Start-I/O routine
FUNCTB=XX_FUNCTABLE,-               ;FDT address
CANCEL=IOC_STD$CANCELIO,-               ;Cancel-I/O routine
REGDMP=XX_REGDUMP,-                 ;Register dumping routine
DIAGBF=<<15*4>+<<3+5+1>*4>>,-       ;Diagnostic buffer size
ERLGBF=<<15*4>+<1*4>+<EMB$L_DV_REGSAV>>  ;Error message buffer size
```

This code excerpt uses the DDTAB macro to create a driver dispatch table for the XX device type.

Notes for Converting Step 1 Drivers

- The OpenVMS AXP version of the DDTAB macro does not automatically define the code psect $$$115_DRIVER; rather, it invokes the DRIVER_DATA macro to include the DDT in data psect $$$110_DATA. On OpenVMS AXP systems, you must explicitly invoke the DRIVER_CODE macro to define the $$$115_DRIVER code psect prior to the first line of executable code.
- The number and order of the arguments to the DDTAB macro are different on OpenVMS AXP systems than on OpenVMS VAX systems.
- On OpenVMS AXP systems, you do not distinguish (by use of the plus sign (+)) entry points of OpenVMS routines that are at absolute addresses from entry points at relative locations within the driver. For instance, an OpenVMS AXP device driver could specify the following argument to the DDTAB macro:

  ```
  CANCEL=IOC_STD$CANCELIO,-
  ```

  It is the equivalent of the following argument specification in an OpenVMS VAX device driver:

  ```
  CANCEL=+IOC$CANCELIO,-
  ```

- An OpenVMS AXP device driver that employs the kernel process services specifies the name of EXE_STD$KP_STARTIO in **start** argument, and the procedure value of the driver's start-I/O routine in the **kp_startio** argument.
- An OpenVMS AXP device driver that employs the kernel process services indicates the size of the kernel mode stack in the **kp_stack_size**, and specifies a mask of registers to be preserved across kernel process context switches in the **kp_reg_mask** argument.
- Because the procedure value of the controller initialization routine is stored in the DDT (DDT$PS_CTRLINIT) in OpenVMS AXP systems, you specify its location by using the new **ctrlinit** argument to the DDTAB macro. (On OpenVMS VAX systems, you specify the location of the controller initialization by issuing a DPT_STORE macro to VEC$L_INITIAL.)
- The OpenVMS AXP version of the DDTAB macro does not provide the **unsolic** argument.
- Although the **channel_assign**, **cancel_selective**, **aux_storage**, and **aux_routine** arguments are allowed in the macro invocation, the functionality they represent has not yet been implemented in OpenVMS AXP systems.
- An OpenVMS AXP terminal port driver cannot share a single DDT with the OpenVMS AXP terminal class driver. The terminal port driver must invoke the DDTAB macro specifying the **ctrlinit** and **unitinit** arguments. The CLASS_UNIT_INIT macro, when invoked by the port driver, initializes the remainder of the port driver's DDT from the class driver's DDT.

♦

DPTAB

Generates a driver prologue table (DPT) in a program section called $$$105_PROLOGUE.

Format

```
DPTAB  [end] ,adapter ,[flags=0] ,ucbsize ,[unload] ,[maxunits=8]
       ,[defunits=1] ,[deliver] ,[vector] [,name] ,[smp=NO] ,[decode] ,step=0,
       [,idb_crams=0] [,ucb_crams=0] [,bt_order] [,ddt=DDT$BASE]
       [,struc_init=DRIVER$STRUC_INIT] [,struc_reinit=DRIVER$STRUC_REINIT]
       [,psect=$$$105_PROLOGUE] [,dpt=DRIVER$DPT]
```

Parameters

end

Unused in OpenVMS AXP device drivers.

adapter

Type of adapter. You can supply any name that, when appended to the string "AT$_", results in a symbol defined by the $DCDEF macro in SYS$LIBRARY:STARLET.MLB. Of these symbols, the driver-loading procedure takes special action only when the keyword **NULL** is present. The driver-loading procedure creates no ADP for a null adapter (AT$_NULL) and clears the VEC$PS_ADP and IDB$L_ADP fields.

[flags]

Flags used in loading the driver. Drivers use the following flags:

DPT$M_SVP — Indicates that the driver requires a permanently allocated system page. Disk drivers use this SPTE during ECC correction and when using the system routines IOC_STD$MOVFRUSER and IOC_STD$MOVTOUSER.

When this flag is set, the driver-loading procedure allocates a permanent system page-table entry (SPTE) for the device. It stores an index to the virtual address of the SPTE in UCB$L_SVPN when it creates the UCB. A driver can calculate the system virtual address of the page corresponding to this index by using the following formula:

> SVA = SEXT((LEFT_SHIFT(ucbl_svpn, mmggl_vpn_to_va)) OR va$m_system)

DPT$M_NOUNLOAD — Indicates that the driver cannot be reloaded. When this bit is set, the driver can be unloaded only by rebooting the system. Note that driver unloading and reloading are not supported on OpenVMS AXP systems.

DPT$M_SMPMOD	Indicates that the driver has been designed to execute within an OpenVMS multiprocessing environment. Use of any of the multiprocessing synchronization macros (DEVICELOCK/DEVICEUNLOCK, FORKLOCK/FORKUNLOCK, or LOCK/UNLOCK) automatically sets this flag, as long as the code using the macro resides in the same module as the invocation of DPTAB.
DPT$M_DECW_DECODE	Indicates that the driver is a DECwindows class input (decoding) driver
DPT$M_NO_IDB_DISPATCH	Tells the driver-loading procedure not to create a list of UCB addresses at the end of the IDB (at IDB$L_UCBLST), regardless of the value of the **maxunits** argument or the maximum units specified in the /MAX_UNITS qualifier of the System Management (SYSMAN) utility command IO CONNECT.

ucbsize

Size in bytes of each UCB the driver-loading procedure creates for devices supported by the driver. This required argument allows drivers to extend the UCB to store device-dependent data describing an I/O operation.

[unload]

Address of the driver routine invoked by the driver-loading procedure before it unloads an old version of the driver to load a new version.

Note

The OpenVMS AXP operating system does not yet permit driver reloading and does not support driver-unloading routines.

[maxunits=8]

Maximum number of units that this driver supports on a controller. If you omit the **maxunits** argument, the default is eight units. You can override the value specified in the DPT at driver-loading time by using the /MAX_UNITS qualifier to the SYSMAN command IO CONNECT. If DPT$M_NO_IDB_DISPATCH is not specified in the **flags** argument to the DPTAB macro, these values affect the size of the UCB list the driver-loading procedure generates at the end of the IDB.

[defunits=1]

Maximum number of UCBs to be created by the autoconfiguration facility (one for each device unit to be configured). The unit numbers assigned are zero to **defunits**–1.

If you do not specify the **deliver** argument, the autoconfiguration facility creates the number of units specified by **defunits**. If you specify the address of a unit delivery routine in the **deliver** argument, the autoconfiguration facility calls that routine to determine whether to create each UCB automatically.

[deliver]

Address of the driver unit delivery routine. The unit delivery routine determines which device units supported by this driver the autoconfiguration facility should configure automatically. If you omit the **deliver** argument, the autoconfiguration facility creates the number of units specified by the **defunits** argument.

[vector]
Address of a driver-specific transfer vector. A terminal port driver specifies the address of its vector table in this argument.

[name]
Name of the device driver. Because the OpenVMS AXP driver-loading procedure automatically generates a driver name and writes it to the DPT, it effectively ignores this argument.

[smp=NO]
Indication of whether the driver is suitably synchronized to execute in an OpenVMS multiprocessing system. Note that use of any of the spin lock synchronization macros in a device driver causes the DPTAB macro to indicate multiprocessing synchronization. All OpenVMS AXP drivers must specify **smp=YES**.

[decode]
Address of counted ASCII string that identifies a DECwindows class input (decoding) driver to serial-line switching code.

step
OpenVMS AXP driver step number. You must indicate that a given driver conforms to the coding practices for an Step 1 OpenVMS AXP device driver by supplying **step=2** in the DPTAB macro invocation. If you specify **step**=1, the macro generates the following message:

```
%MACRO-E-GENERR, Generated ERROR: *CAUTION* Step 1 drivers will be obsolete in V2.0
```

If you omit the **step** argument entirely, or specify a value other than 1 or 2, the DPTAB macro generates the message:

```
%MACRO-E-GENERR, Generated ERROR: DPTAB must declare driver STEP=1 or STEP=2
```

Step 2 drivers may also optionally supply a value for the **step** argument of the DDTAB macro. If the step values given the DPTAB and DDTAB macros conflict, the DPTAB macro generates an error of the form:

```
%MACRO-E-GENERR, Generated ERROR: DPTAB STEP=x conflicts with prior declaration.
```

idb_crams
Number of CRAMS to be allocated and associated with the IDB. The driver-loading procedure allocates the number of CRAMs specified in **idb_crams** argument to the DPTAB macro and inserts them in the linked list headed by IDB$PS_CRAM. These CRAMs are therefore available to the driver's controller and unit initialization routine.

ucb_crams
Number of CRAMS to be allocated and associated with the UCB. The driver-loading procedure allocates the number of CRAMs specified in **ucb_crams** argument to the DPTAB macro and inserts them in the linked list headed by UCB$PS_CRAM. These CRAMs are therefore available to the driver's unit initialization routine.

[bt_order]
Ordering number for call to the runtime drivers for boot devices.

[ddt=DDT$BASE]
Address of DDT. The default is required for all non-Digital-supplied device drivers.

[struc_init=DRIVER$STRUC_INIT]
Address of the driver I/O database initialization routine automatically generated by an invocation of the DPT_STORE macro with the **INIT** label. This routine initializes those data structure fields indicated by the invocations of the DPT_STORE macro that follow the DPT_STORE **INIT** and precede the DPT_STORE **REINIT**. The driver-loading procedure calls this initialization routine when it creates the structures and loads the driver, prior to calling the driver's controller and unit initialization routines.

The default value of this argument is required for all Step 2 OpenVMS AXP device drivers.

[struc_reinit=DRIVER$STRUC_REINIT]
Address of the driver I/O database reinitialization routine automatically generated by an invocation of the DPT_STORE macro with the **REINIT** label. This routine initializes those data structure fields indicated by the invocations of the DPT_STORE macro that follow the DPT_STORE **INIT** and precede the DPT_STORE **END**. The driver-loading procedure calls this reinitialization routine when the driver is first loaded into the system, and whenever the driver is reloaded, prior to calling the driver's controller and unit initialization routines.

The default value of this argument is required for all Step 2 OpenVMS AXP device drivers.

Note that driver unloading and reloading are not supported on OpenVMS AXP systems.

[psect=$$$105_PROLOGUE]
Program section in which the DPT is created. The default value of this argument is required for all non-Digital-supplied device drivers.

[,dpt=DRIVER$DPT]
Global symbol for DPT location. The default value of this argument is required for all non-Digital-supplied device drivers.

Description

The DPTAB macro, in conjunction with invocations of the DPT_STORE macro, creates a driver prologue table (DPT). The DPTAB macro places information in the DPT that allows the driver-loading procedure to identify the driver and the devices it supports. The DPTAB macro, in invoking the $SPLCODDEF definition macro, also defines the spin lock indexes used in the DPT_STORE, FORKLOCK, and LOCK macros.

Example

```
DPTAB   STEP=2,-                ;OpenVMS AXP Step 2 driver
        ADAPTER=CI,-            ;Adapter type
        UCBSIZE=UCB$C_PASIZE,-  ;UCB size
        NAME=PNDRIVER,-         ;Driver name
        SMP=YES,-               ;SMP capable
        FLAGS=<DPT$M_SCS!-      ;Driver requires SCS load,
                DPT$M_NOUNLOAD> ; cannot be reloaded
DPT_STORE INIT
DPT_STORE       UCB,UCB$B_FLCK,B,SPL$C_SCS ;SCS spinlock
DPT_STORE       UCB,UCB$L_DEVCHAR,L,<-  ;Device characteristics:
                DEV$M_SHR!-             ; Sharable
                DEV$M_AVL!-             ; Available
                DEV$M_ELG!-             ; Error logging device
                DEV$M_IDV!-             ; Input device
                DEV$M_ODV>              ; Output device
DPT_STORE       UCB,UCB$B_DIPL,B,PN_BR_LEVEL+16 ;Device interrupt IPL
DPT_STORE       UCB,UCB$B_DEVCLASS,B,-  ;Device class =
                DC$_BUS                 ; bus
DPT_STORE       UCB,UCB$L_ERTMAX,L,50   ;Retry count is 50 times
DPT_STORE       UCB,UCB$L_ERTCNT,L,50   ; without reboot of system
DPT_STORE REINIT
DPT_STORE       DDB,DDB$L_DDT,D,PN$DDT  ;DDT address
DPT_STORE_ISR   CRB$L_INTD,PN$MISC_INTERRUPT ; ISR address
DPT_STORE_ISR   CRB$L_INTD+<CRB$S_INTD>,PN$RSP_INTERRUPT
DPT_STORE       END
```

This excerpt from PNDRIVER.MAR contains the DPTAB macro and the series of DPT_STORE and DPT_STORE macros that create its driver prologue table.

Notes for Converting Step 1 Drivers

- You must indicate that a given driver conforms to the coding practices for a Step 2 OpenVMS AXP device driver by supplying **step=2** in the **step** argument.
- A driver can request the driver-loading procedure to allocate CRAMs and associate them with the IDB or UCB by specifying the **idb_crams** and **ucb_crams** arguments.
- The OpenVMS AXP driver-loading procedure does not support the reloading of Step 1 OpenVMS AXP device drivers. It therefore ignores the **unload** argument to the DPTAB macro.
- Because the OpenVMS AXP driver-loading procedure automatically generates a driver name and writes it to the DPT, it effectively ignores the **name** argument.
- OpenVMS AXP ignores the **end** argument.
- The DPTAB macro, in conjunction with invocations of the DPT_STORE macro which specify the **INIT**, **REINIT**, and **END** labels, automatically generates driver structure initialization and reinitialization routines, storing their procedure values in the DPT. The default global symbol name fot the DPT location is now DRIVER$DPT instead of EVMS$DRIVER_DPT.

♦

DPT_STORE

In the context of a DPTAB macro invocation, generates driver structure initialization and reinitialization routines which the driver loading and reloading procedures call to store values in a table or data structure.

Format

DPT_STORE   str_type ,str_off ,oper ,exp [,pos] [,size]

Parameters

str_type

Type of data structure (CRB, DDB, IDB, ORB, or UCB) into which the driver-loading procedure is to store the specified data, or a label denoting a table marker. Table marker labels indicate the start of a list of DPT_STORE macro invocations that store information for the driver-loading procedure in the driver initialization table and driver reinitialization table sections of the DPT. If this argument is a table marker label, no other argument is allowed. The following labels are used:

INIT	Indicates the start of fields to initialize when the driver is loaded
REINIT	Indicates the start of additional fields to initialize when the driver is loaded and reinitialized when the driver is reloaded
END	Indicates the end of the two lists

str_off

Unsigned offset into the data structure in which the data is to be stored. This value cannot be more than 65,535 bytes.

oper

Type of storage operation, one of the following:

Type	Meaning
B	Write a byte value.
W	Write a word value.
L	Write a longword value.
D	Write an address relative to the beginning of the driver.
V	Write a bit field. If you specify a **V** in the **oper** argument, the driver-loading procedure uses the **exp**, **pos**, and **size** arguments in the bit insertion operation.

If an at sign (@) precedes the **oper** argument, the **exp** argument indicates the address of the data that is to be stored and not the data itself.

exp

Expression indicating the value with which the driver-loading procedure is to initialize the indicated field. If an at sign character (@) precedes the **oper** argument, the **exp** argument indicates the address of the data with which to initialize the field. For example, the following macro indicates that the contents of the location DEVICE_CHARS are to be written into the DEVCHAR field of the UCB.

```
DPT_STORE UCB,UCB$L_DEVCHAR,@L,DEVICE_CHARS
```

[pos]
Starting bit position within the specified field; used only if **oper=V**.

[size]
Number of bits to be written; used only if **oper=V**.

Description

The DPT_STORE macro provides a mechanism for a driver to initialize specific data structure fields when the driver is first loaded and when the driver is reloaded. A driver typically contains a series of DPT_STORE invocations which, together, automatically create a driver I/O database initialization routine and a driver I/O database reinitialization routine. The DPTAB macro writes the locations of these routines in the DPT. The driver-loading routine calls the initialization routine when a driver is first loaded; it calls the reinitialization routine both when the driver is first loaded and when the driver is reloaded. [Note that Step 2 OpenVMS AXP device drivers cannot be reloaded.]

A driver constructs the initialization tables by following the DPTAB macro with one or more invocations of the DPT_STORE macro.

Drivers use the DPT_STORE macro with the **INIT** table marker label to begin a list of DPT_STORE invocations that supply initialization data for the following fields:

UCB$B_FLCK — Index of the fork lock under which the driver performs fork processing. Fork lock indexes are defined by the $SPLCODDEF definition macro (invoked by DPTAB) as follows:

IPL	Fork Lock Index
8	SPL$C_IOLOCK8
9	SPL$C_IOLOCK9
10	SPL$C_IOLOCK10
11	SPL$C_IOLOCK11

UCB$B_DIPL — Device interrupt priority level

Other commonly initialized fields are as follows:

UCB$L_DEVCHAR	Device characteristics
UCB$B_DEVCLASS	Device class
UCB$B_DEVTYPE	Device type
UCB$W_DEVBUFSIZ	Default buffer size
UCB$Q_DEVDEPEND	Device-dependent parameters

Drivers use the DPT_STORE macro with the **REINIT** table marker label to begin a list of DPT_STORE and DPT_STORE_ISR invocations that supply initialization and reinitialization data. The following fields are declared with the DPT_STORE_ISR macro:

CRB$L_INTD	Interrupt service routine
CRB$L_INTD2	Interrupt service routine for second interrupt vector

For an example of the use of the DPT_STORE macro, see the description of the DPTAB macro.

Notes for Converting Step 1 Drivers

- Because the OpenVMS AXP driver-loading procedure automatically stores the address of the DDT in the DDB, an OpenVMS AXP device driver does not invoke the DPT_STORE macro to write this address. For instance, the following line should be removed from an existing OpenVMS VAX driver that is to be moved to OpenVMS AXP:

```
DPT_STORE  DDB,DDB$L_DDT,D,XA$DDT  ;Address of DDT
```

- Because the procedure value of the controller and unit initialization routines are stored in the DDT (DDT$PS_CTRLINIT and DDT$L_UNITINIT, respectively) in OpenVMS AXP systems, you specify their location by using the **ctrlinit** and **unitinit** arguments to the DDTAB macro. The following uses of the DPT_STORE macro do not work on OpenVMS AXP systems:

```
DPT_STORE  CRB,VEC$L_INITIAL,D,XA$CTRL_INIT  ;Address of controller init routine
DPT_STORE  CRB,VEC$L_UNITINIT,D,XA$UNIT_INIT ;Address of unit init routine
```

- Because the interrupt dispatcher requires the addresses of both the code entry point and the procedure descriptor of an interrupt service routine, you must use the new DPT_STORE_ISR macro (which generates both) to declare the routine. For instance, you should use:

```
DPT_STORE_ISR   CRB$L_INTD, XA_INTERRUPT
                                ;Address of interrupt service routine
```

  instead of:

```
DPT_STORE  CRB,CRB$L_INTD+VEC$L_ISR,D,-
              XA_INTERRUPT      ;Address of interrupt service routine
```

- The DPTAB macro, in conjunction with invocations of the DPT_STORE macro which specify the **INIT**, **REINIT**, and **END** labels, automatically generates driver structure initialization and reinitialization routines, storing their procedure values in the DPT.

- Be aware that certain data structure fields (such as UCB$B_FIPL) have been made obsolete in OpenVMS AXP. The names of other fields may have changed, typically to reflect a change in size of the datum.

♦

DPT_STORE_ISR

In the context of a DPTAB macro invocation, generates the addresses of the code entry point and procedure descriptor of an interrupt service routine and stores them in the interrupt transfer vector block (VEC).

Format

DPT_STORE_ISR vec_off ,entry

Parameters

vec_off
Symbolic offset to interrupt transfer vector within the CRB. These offsets are of the following form:

Symbolic Offset	Description
CRB$L_INTD	First interrupt transfer vector
CRB$L_INTD2	Second interrupt transfer vector
CRB$L_INTD+<2*VEC$K_LENGTH>	Third interrupt transfer vector

entry
Procedure value of an interrupt service routine.

Description

The DPT_STORE_ISR macro provides a mechanism for a driver to initialize the VEC$PS_ISR_PD and VEC$PS_ISR_CODE fields of an interrupt transfer vector block (VEC) with the addresses of an interrupt service routine's procedure descriptor and code entry point, respectively. Like invocations of the DPT_STORE macro, you invoke the DPT_STORE_ISR macro within the context of the DPTAB macro.

Typically, you use DPT_STORE_ISR within the reinitialization section of the DPT (following DPT_STORE REINIT), so that the VEC fields are initialized at both driver loading and reloading.

Example

```
DPT_STORE_ISR   CRB$L_INTD, XA_INTERRUPT
```

This invocation of the DPT_STORE_ISR macro locates the first interrupt transfer vector associated with the device controller, and places the address of XA_INTERRUPT's procedure descriptor in VEC$PS_ISR_PD and the address of its code entry point in VEC$PS_ISR_CODE.

FDT_ACT

Initializes the FDT action routine vector slot corresponding to one or more specified I/O function codes with the procedure value of the specified upper-level FDT action routine.

Format

FDT_ACT   action, codes

Parameters

action

Action routine that services the I/O function codes identified by the **codes** argument.

codes

List of codes (enclosed within angle brackets and separated by commas) for I/O functions serviced by the specified upper-level FDT action routine. The macro expansion prefixes each code with the string IO$_; for example, READVBLK expands to IO$_READVBLK.

Description

The FDT_ACT macro identifies the upper-level FDT action routine that processes one or more specified I/O function codes. If, at the time it invokes FDT_ACT, the driver has not yet invoked the FDT_INI macro, FDT_ACT invokes it on the driver's behalf, creating an FDT with the label DRIVER$FDT.

An OpenVMS AXP device driver specifies one or more legal I/O functions by supplying the address of an upper-level FDT action routine for that function to the FDT_ACT macro. The FDT_ACT macro initializes the slot in the FDT action routine vector corresponding to each supplied function code with the procedure value of the specified routine.

Multiple invocations of the FDT_ACT macro, in sum, define the full set of I/O functions serviced by the driver. An illegal I/O function is one that the driver does not list in any FDT_ACT macro invocations. Its vector slot contains the procedure value of the illegal I/O function processing routine (EXE$ILLIOFUNC).

Note, however, only one upper-level FDT action routine can service any given I/O function. If you reuse an I/O function code in an FDT_ACT invocation, the compiler generates an error of the form:

```
%MACRO-E-GENERR, Generated ERROR: Multiple actions defined for function IO$_xxxxxx
```

A consequence of this limitation is that, if the preprocessing of a given function requires that several routines be executed, the upper-level FDT action routine must set up the appropriate call chain.

Example

```
XX_FUNCTABLE:                              ;Function decision table
     FDT_INI   XX$FDT
     FDT_BUF   -                           ;Buffered-I/O functions
               <READLBLK,-                 ;Read logical block
                READPBLK,-                 ;Read physical block
                READVBLK,-                 ;Read virtual block
                SENSEMODE,-                ;Sense reader mode
                SENSECHAR,-                ;Sense reader characteristics
                SETMODE,-                  ;Set reader mode
                SETCHAR,-                  ;Set reader characteristics
               >
     FDT_ACT   XX_READ,-                   ;Read function FDT routine
               <READLBLK,-                 ;Read logical block
                READPBLK,-                 ;Read physical block
                READVBLK,-                 ;Read virtual block
               >
     FDT_ACT   EXE_STD$SETMODE,-           ;Set mode/characteristics FDT routine
               <SETCHAR,-                  ;Set reader characteristics
                SETMODE,-                  ;Set reader mode
               >
     FDT_ACT   EXE_STD$SENSEMODE,-         ;Sense mode/characteristics FDT routine
               <SENSECHAR,-                ;Sense reader characteristics
                SENSEMODE,-                ;Sense reader mode
               >
```

This function decision table specifies that the routine XX_READ be called for all read functions that are valid for the device. XX_READ appears later in the driver module. System I/O preprocessing will call routines EXE_STD$SETMODE and EXE_STD$SENSEMODE for the device's set-characteristics and sense-mode functions.

FDT_BUF

Builds the buffered function mask within a driver's function decision table (FDT) from the specified list of I/O functions.

Format

FDT_BUF [codes]

Parameters

[codes]
List of codes (enclosed within angle brackets and separated by commas) for I/O functions supported by the driver that require an intermediate system buffer. The macro expansion prefixes each code with the string IO$_; for example, READVBLK expands to IO$_READVBLK.

Description

The FDT_BUF macro builds the buffered function mask within an FDT from the specified list of I/O functions.

An OpenVMS AXP device driver invokes the FDT_BUF macro to indicate which of the I/O functions it supports require a system buffer. If the driver has not yet invoked the FDT_INI macro, FDT_BUF invokes it on the driver's behalf, creating an FDT with the label DRIVER$FDT.

A driver specifies a legal I/O function by supplying the address of an upper-level FDT action routine for that function to the FDT_ACT macro. Beware of specifying a function code in an FDT_BUF invocation that you do not also specify in an FDT_ACT invocation. The FDT action routine vector slot for such a function contains a pointer to the illegal I/O function processing routine (EXE$ILLIOFUNC).

An example of the use of FDT_BUF appears in the description of the FDT_ACT macro.

FDT_INI

Creates, labels, and initializes a function decision table (FDT).

Format

FDT_INI [fdt=DRIVER$FDT]

Parameters

[fdt=DRIVER$FDT]
Label of the start of the FDT.

Description

The FDT_INI macro creates an FDT, using the DRIVER_DATA macro to place it within the driver's data program section ($$$110_DATA). The macro properly aligns the FDT in memory, assigning it the label specified by the **fdt** argument.

FDT_INI initializes the FDT by clearing the buffered function mask and entering the address of the illegal I/O function processing routine (EXE$ILLIOFUNC) in all FDT action routine vector slots.

An OpenVMS AXP device driver invokes the FDT_BUF macro to indicate which of the I/O functions it supports require a system buffer. A driver specifies a legal I/O function by supplying the address of an upper-level FDT action routine for that function to the FDT_ACT macro.

An example of the use of FDT_INI appears in the description of the FDT_ACT macro.

FORK

Creates a simple fork process on the local processor.

Format

FORK [routine] [,continue] [environment=JSB | CALL]

Parameters

[routine]
Name of the routine to be executed in fork context. If you omit this argument, the FORK macro assumes that the fork routine immediately follows the invocation.

[,continue]
Label where execution continues after the fork block has been inserted on the fork queue. If you omit this argument, control returns to the caller of the routine that invoked the FORK macro.

[,environment
Keyword that specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then EXE$PRIMITIVE_FORK is called and a .JSB_ENTRY directive is used to generate the fork routine. If specified as CALL, then EXE_STD$PRIMITIVE_FORK is called, a .CALL_ENTRY directive is used to generate the fork routine, the FR3, FR4, and FKB parameters in the fork routine are copied into R3, R4, and R5.

Description

The FORK macro creates a fork process. When the FORK macro is invoked, the following registers must contain the values listed:

Register	Contents
R3	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR3(R5)
R4	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR4(R5)
R5	Contains a pointer to the fork block

Unlike the IOFORK macro, the FORK macro does not disable device timeouts by clearing the UCB$V_TIM bit in the field UCB$L_STS.

Notes for Converting Step 1 Drivers

```
Implicit outputs to caller:

  ENVIRONMENT=CALL

    R0,R1        are scratched.

  ENVIRONMENT=JSB

    R3,R4        outputs from EXE$PRIMITIVE_FORK.

    R0,R1        are preserved.

Implicit outputs to fork routine, i.e. entry conditions:

  ROUTINE=routine_name

                 If the routine name is specified  then  the  fork
                 entry  point  is  assumed  to  be  at  the  named
                 location and no fork entry point is defined here.
                 The  named  fork  routine  can use either the new
                 standard call interface or  the  traditional  JSB
                 interface  as described in section 4.2 regardless
                 of the setting of the ENVIRONMENT keyword.

  ROUTINE=<not specified>,ENVIRONMENT=CALL

                 A fork routine entry point  is  generated  for  a
                 routine  using the new standard call interface as
                 described in section 4.2.

    R3,R4,R5     contain traditional fork routine parameter values
                 copied  from  the  standard call interface actual
                 parameters,

    R0,R1        can be scratched.

  ROUTINE=<not specified>,ENVIRONMENT=JSB

                 A fork routine entry point  is  generated  for  a
                 routine  using  the  traditional JSB interface as
                 described in section 4.2.

    R3,R4,R5     contain traditional fork routine parameters,

    R0-R4        can be scratched.
```

♦

FORK_ROUTINE

Defines the entry point of a fork routine.

Format

FORK_ROUTINE [name=fork_routine_name] [,symbol=LOCAL | GLOBAL] [,environment=JSB | CALL] [,fetch=YES | NO]

Parameters

[name]
Name of the fork routine.

[,symbol]
Specifies if the routine name should be declared as a local or global symbol. The default is for a local symbol.

[,environment]
Specifies the fork routine environment as either JSB or CALL. If specified as JSB, then a .JSB_ENTRY directive is used to define the fork routine entry point. If specified as CALL, then a .CALL_ENTRY directive is used to define the fork routine entry point. The default is JSB.

[,fetch]
Specifies if the fork routine parameters for an ENVIRONMENT=CALL fork routine should be copied into the traditional R3. R4. and R5 register The default is YES.

Notes for Converting Step 1 Drivers

```
Implicit inputs:

  None.

Implicit outputs, i.e.  fork routine entry conditions:

  ENVIRONMENT=CALL

    FORKARG$_FR3(AP), FORKARG$_FR4(AP), FORKARG$_FKB(AP)
                 the symbolic parameter offsets  are  defined  and
                 can   be   used   to   access  the  fork  routine
                 parameters,

    R3,R4,R5     contain traditional fork  routine  parameters  if
                 FETCH=YES,

    R0,R1        can be scratched.

  ENVIRONMENT=JSB

    R3,R4,R5     contain traditional fork routine parameters,

    R0-R4        can be scratched.
```

♦

FORK_WAIT

Inserts a fork block on the fork-and-wait queue.

Format

FORK_WAIT [routine] [,continue] [,environment=JSB | CALL]

Parameters

[routine]
Name of the routine to be executed in fork context. If you omit this argument, the FORK_WAIT macro assumes that the fork routine immediately follows the invocation.

[,continue]
Label where execution continues after the fork block has been inserted on the fork-and-wait queue. If you omit this argument, control returns to the caller of the routine that invoked the FORK_WAIT macro.

[,environment]
Specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then EXE$PRIMITIVE_FORK_WAIT is called and a .JSB_ENTRY directive is used to generate the fork routine. If specified as CALL, then EXE_STD$PRIMITIVE_FORK_WAIT is called, a .CALL_ENTRY directive is used to generate the fork routine, the FR3, FR4, and FKB parameters in the fork routine are copied into R3, R4, and R5.

Description

The FORK_WAIT macro inserts a fork block on the system fork-and-wait queue. When the FORK_WAIT macro is invoked, the following registers must contain the values listed:

Register	Contents
R3	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR3(R5)
R4	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR4(R5)
R5	Contains a pointer to the fork block

Notes for Converting Step 1 Drivers

```
Implicit outputs to caller:

  ENVIRONMENT=CALL

    R0,R1        are scratched.

  ENVIRONMENT=JSB

    R0,R1        are preserved.

Implicit outputs to fork routine, i.e. entry conditions:

  ROUTINE=routine_name

                 If the routine name is specified  then  the  fork
                 entry  point  is  assumed  to  be  at  the  named
                 location and no fork entry point is defined here.
                 The  named  fork  routine  can use either the new
                 standard call interface or  the  traditional  JSB
                 interface  as described in section 4.2 regardless
                 of the setting of the ENVIRONMENT keyword.

  ROUTINE=<not specified>,ENVIRONMENT=CALL

                 A fork routine entry point  is  generated  for  a
                 routine  using the new standard call interface as
                 described in section 4.2.

    R3,R4,R5     contain traditional fork routine parameter values
                 copied  from  the  standard call interface actual
                 parameters,

    R0,R1        can be scratched.

  ROUTINE=<not specified>,ENVIRONMENT=JSB
                 A fork routine entry point  is  generated  for  a
                 routine  using  the  traditional JSB interface as
                 described in section 4.2.

    R3,R4,R5     contain traditional fork routine parameters,

    R0-R4        can be scratched.
```

♦

IOFORK

Creates a fork process on the local processor for a device driver, disabling timeouts from the associated device.

Format

IOFORK [routine] [,continue] [,ENVIRONMENT=JSB|CALL

Parameters

[routine]
Name of the routine to be executed in fork context. If you omit this argument, the IOFORK macro assumes that the fork routine immediately follows the invocation.

[,continue]
Label where execution continues after the fork block has been inserted on the fork queue. If you omit this argument, control returns to the caller of the routine that invoked the IOFORK macro.

[,environment
Keyword that specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then EXE$PRIMITIVE_FORK is called and a .JSB_ENTRY directive is used to generate the fork routine. If specified as CALL, then EXE_STD$PRIMITIVE_FORK is called, a .CALL_ENTRY directive is used to generate the fork routine, the FR3, FR4, and FKB parameters in the fork routine are copied into R3, R4, and R5.

Description

The IOFORK macro disables device timeouts by clearing the UCB$V_TIM bit in the field UCB$L_STS and creates a fork process. When the IOFORK macro is invoked, the following registers must contain the values listed:

Register	Contents
R3	Contents to be placed in R3 of the fork process (64 bits)
R4	Contents to be placed in R4 of the fork process (64 bits)
R5	Address of fork block

Notes for Converting Step 1 Drivers

```
Implicit outputs to caller:
  ENVIRONMENT=CALL
    R0,R1        are scratched.
  ENVIRONMENT=JSB
    R3,R4        outputs from EXE$PRIMITIVE_FORK.
    R0,R1        are preserved.
Implicit outputs to fork routine, i.e. entry conditions:
```

ROUTINE=routine_name

If the routine name is specified then the fork entry point is assumed to be at the named location and no fork entry point is defined here. The named fork routine can use either the new standard call interface or the traditional JSB interface as described in section 4.2 regardless of the setting of the ENVIRONMENT keyword.

ROUTINE=<not specified>,ENVIRONMENT=CALL

A fork routine entry point is generated for a routine using the new standard call interface as described in section 4.2.

R3,R4,R5 contain traditional fork routine parameter values copied from the standard call interface actual parameters,

R0,R1 can be scratched.

ROUTINE=<not specified>,ENVIRONMENT=JSB

A fork routine entry point is generated for a routine using the traditional JSB interface as described in section 4.2.

R3,R4,R5 contain traditional fork routine parameters,

R0-R4 can be scratched.

♦

RELCHAN

Releases all controller data channels allocated to a device.

Format

RELCHAN

Description

The RELCHAN macro releases all controller data channels allocated to a device. When the RELCHAN macro is invoked, R5 must contain the address of the UCB. RELCHAN destroys the contents of R0 through R1.

REQCHAN

Requests exclusive use of the CRB and defines the channel grant routine entry point.

Format

REQCHAN [pri=LOW | HIGH] [,ENVIRONMENT=JSB | CALL)

Parameters

[pri=LOW]
Priority of request. If the priority is **HIGH**, REQCHAN calls IOC_STD$PRIMITIVE_REQCHANH; otherwise it calls IOC_STD$PRIMITIVE_REQCHANL.

[,environment]
Specifies the callers and grant routine environments as either JSB or CALL. The default is JSB. If specified as JSB, then an RSB is used to return from the current routine if the channel is not granted immediately and a .JSB_ENTRY directive is used to generate the grant routine. If specified as CALL, then an RET is used to return from the current routine if the channel is not granted immediately, a .CALL_ENTRY directive is used to generate the grant routine, and the grant routine parameters are copied into R3, R4, and R5.

Description

The REQCHAN macro obtains a controller's data channel.

If the channel is granted immediately, execution continues at the line of code that immediately follows the macro invocation. If no channel is available, the UCB is placed in a channel-wait queue, and the macro returns control to its caller's caller. When the channel request is granted, execution resumes at the line of code following the macro execution.

When the REQCHAN macro is invoked, R5 must contain the address of the UCB.

The REQCHAN macro returns the address of the IDB in R4 and destroys the contents of R0 through R2.

Notes for Converting Step 1 Drivers

Implicit inputs: R3 contains a pointer to the IRP which if necessary is passed to the grant routine via UCB$Q_FR3(R5), R5 contains a pointer to the UCB, Implicit outputs to caller: R4 contains the IDB address, R0-R2 are scratched. Implicit outputs to grant routine, i.e. entry conditions: ENVIRONMENT=CALL A driver channel grant routine entry point is generated for a routine using the new standard call interface as described in section 4.3. R3,R4,R5 contain traditional channel grant routine parameter values copied from the standard call interface actual parameters, R0,R1 can be scratched. ENVIRONMENT=JSB A driver channel grant routine entry point is generated for a routine using the traditional JSB interface as described in section 4.3. R3,R4,R5 contain traditional channel grant routine parameters, R0-R5 can be scratched.

♦

REQCOM

Places the current IRP on the post processing queue and to logically end the driver fork thread that began on entry into the start I/O or alternate start I/O routines.

Format

REQCOM [,environment=JSB | CALL]

Parameters

[,environment]
Specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then an RSB is used to return from the current routine. If specified as CALL, then an RET is used to return from the current routine.

Description

The REQCOM macro completes the processing of an I/O request after the driver has finished its portion of the processing.

When the REQCOM macro is invoked, the following registers must contain the following values:

Register	Contents
R0	First longword of I/O status
R1	Second longword of I/O status
R5	Address of UCB

The REQCOM macro destroys the contents of R0 through R1. All other registers are also destroyed if the action of the macro initiates the processing of a waiting I/O request for the device.

REQPCHAN

Obtains a controller's data channel.

Format

REQPCHAN [pri=LOW] [,environment=JSB | CALL]

Parameters

[pri=LOW]
Priority of request. If the priority is **HIGH**, REQPCHAN calls IOC$PRIMITIVE_REQCHANH; otherwise it calls IOC$PRIMITIVE_REQCHANL.

[,environment=JSB | CALL]
Specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then an RSB is used to return from the current routine. If specified as CALL, then an RET is used to return from the current routine.

Description

The REQPCHAN macro calls obtains a controller's data channel.

If the channel is granted immediately, execution continues at the line of code that immediately follows the macro invocation. If no channel is available, the UCB is placed in a channel-wait queue, and the macro returns control to its caller's caller. When the channel request is granted, execution resumes at the line of code following the macro execution.

When the REQPCHAN macro is invoked, R5 must contain the address of the UCB.

The REQPCHAN macro returns the address of the IDB in R4 and destroys the contents of R0 through R2.

WFIKPCH, WFIRLCH

Suspends a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout. When WFIKPCH is invoked, the fork thread keeps ownership of the controller channel while waiting; when WFIRLCH is invoked, the fork thread releases ownership of the controller channel.

Format

$\left\{ \begin{array}{l} \text{WFIKPCH} \\ \text{WFIRLCH} \end{array} \right\}$ excpt [,time=65536] [,newipl] [environment=JSB | CALL] [,toutrout]

Parameters

excpt
Label of the timeout handling code within the driver.

[time=65536]
Timeout interval, expressed as the number of seconds to wait for an interrupt before a device timeout is considered to exist. A value equal to or greater than *2* is required because the timeout detection mechanism is accurate only to within 1 second.

[newipl=(SP)+]
IPL to which to lower before returning to caller. Typically this is the fork IPL associated with device processing that was pushed on the stack by a prior invocation of the DEVICELOCK macro.

[,environment
Specifies the current and interrupt resume routine environments are either JSB or CALL. The default is JSB. If specified as JSB, then the return from the current procedure is via a RSB, and a .JSB_ENTRY directive is used to generate the resume routine entry point. If specified as CALL, then the return from the current procedure is via a RET, a .CALL_ENTRY directive is used to generate the resume routine entry point, and the IRP, FR4, and UCB parameters in the resume routine are copied into R3, R4, and R5.

[,toutrout]
Specifies the timeout routine entry point. The timeout routine is a fork routine that can either use the traditional or the new standard call interface. If not specified then the resume routine entry point is also used as the timeout routine entry point. The timeout routine procedure value is loaded into UCB$PS_TOUTROUT(R5) for potential use by EXE$TIMEOUT. This parameter can not be specified together with the EXCPT parameter.

Description

The WFIKPCH and WFIRLCH macros construct an inline entry point for the code that follows the macro invocation (normally called the fork routine). They insert an instruction at the beginning of the fork routine that tests UCB$V_TIMOUT in UCB$L_STS and branches to the label of the timeout code (specified in the **excpt** argument) if it is set.

Finally, WFIKPCH and WFIRLCH place the procedure value of the fork routine (at the instruction following the macro invocation) in UCB$L_FPC, insert the **time** value in R1 and **newipl** value in R2, and call the appropriate wait-for-interrupt routine (either IOC$PRIMITIVE_WFIKPCH or IOC$PRIMITIVE_WFIRLCH).

When the wait-for-interrupt routine returns control, the WFIKPCH or WFIRLCH macro issues an RSB instruction to the caller of the routine which invoked it (that is, the caller of the start-I/O routine).

Either the device interrupt servicing routine or the software timer interrupt servicing routine will eventually issue a JSB instruction to the fork routine. In both instances, code can assume that only R3 and R4 have been preserved across the suspension.

IOC$WFIKPCH and IOC$WFIRLCH assume that, prior to the invocation of the macro, a DEVICELOCK macro has been issued to synchronize with other device activity.

When the WFIKPCH or WFIRLCH macro is invoked, the following locations must contain the values listed:

Location	Contents
R5	Address of UCB
00(SP)	IPL at which control is passed to the caller's caller (if the **newipl** argument is not specified)

Notes for Converting Step 1 Drivers

```
Implicit inputs:

  R3            contains a pointer to the IRP which  is  passed  to
                the interrupt resume routine via UCB$Q_FR3(R5),

  R4            contains the 64-bit value to pass to the  interrupt
                resume routine via UCB$Q_FR4(R5),

  R5            contains a pointer to the UCB,

Implicit outputs to caller:

  R0,R1,R2      are scratched.

Implicit outputs to resume routine, i.e. entry conditions:

  ENVIRONMENT=CALL

                An entry point is generated that conforms  to  both
                the new standard call interface for a driver resume
                from interrupt routine as described in section  4.7
                and  the  new  standard  call  interface for a fork
                routine as described in section 4.2.

    R3,R4,R5    contain traditional resume from  interrupt  routine
                parameter  values (64-bit value for R4) copied from
                the standard call interface actual parameters,

    R0,R1       can be scratched.
```

```
ENVIRONMENT=JSB
               An entry point is generated that conforms  to  both
               the  traditional  JSB interface for a driver resume
               from interrupt routine
               and  the  traditional  JSB  interface  for  a  fork
               routine..

  R3,R4,R5     contain traditional resume from  interrupt  routine
               parameters (64-bit value for R4),

  R0-R4        can be scratched.
```

♦

2

System Routines

This chapter describes the operating system routines that are used by device drivers and employs the following conventions:

- Most routines reside in modules within the [SYS] facility of the operating system. A routine description provides a facility name (in brackets) only if the module is not located in the [SYS] facility.
- Many routines are not directly called by device drivers. Rather, the operating system supplies macros that drivers invoke to accomplish the routine call. The description of a routine that has such a macro interface lists the name of the associated macro. Chapter 1 describes how a driver can use these macros.
- System routines generally return a status value in R0 (for instance, SS$_NORMAL). The low-order bit of this value indicates successful (1) or unsuccessful (0) completion of the routine. Additional information on returned status values appears in the *OpenVMS System Services Reference Manual* and the *OpenVMS System Messages and Recovery Procedures Reference Manual*.

ACP_STD$ACCESS

Accesses and creates ACP function processing.

Module

SYSACPFDT

Format

status = ACP_STD$ACCESS (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_DEVNOTMOUNT	Device not mounted.
SS$_DEVFOREIGN	Device is mounted as foreign.
SS$_EXQUOTA	File quota exceeded.
SS$_FILALRACC	File already accessed.

SS$_IVCHNLSEC	Invalid section channel.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$ACCESS as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$ACCESS (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$ACCESS.
- ACP$ACCESS returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$ACCESS returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$ACCESSNET

Connects to network function processing.

Module

SYSACPFDT

Format

status = ACP_STD$ACCESSNET (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.
SS$_EXQUOTA	File quota exceeded.
SS$_FILALRACC	File already accessed.
SS$_IVCHNLSEC	Invalid section channel.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$ACCESSNET as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$ACCESSNET (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$ACCESSNET.
- ACP$ACCESSNET returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$ACCESSNET returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$DEACCESS

Deaccesses ACP function processing.

Module

SYSACPFDT

Format

status = ACP_STD$DEACCESS (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_FILNOTACC	File not accessed.
SS$_IVCHNLSEC	Invalid section channel.
SS$_NORMAL	Normal, successful completion.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$DEACCESS as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$DEACCESS (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$DEACCESS.
- ACP$DEACCESS returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$DEACCESS returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$MODIFY

Deletes and modifies ACP function processing.

Module

SYSACPFDT

Format

status = ACP_STD$MODIFY (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_DEVNOTMOUNT	Device not mounted.
SS$_DEVFOREIGN	Device is mounted as foreign.
SS$_EXQUOTA	File quota exceeded.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$MODIFY as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$MODIFY (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$MODIFY.
- ACP$MODIFY returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$MODIFY returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$MOUNT

Initiates ACP mount function processing.

Module

SYSACPFDT

Format

status = ACP_STD$MOUNT (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_DEVNOTMOUNT	Device not mounted.
SS$_NOPRIV	Process has insufficient privileges.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$MOUNT as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$MOUNT (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$MOUNT.
- ACP$MOUNT returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$MOUNT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$READBLK

Processes a read block ACP function.

Module

SYSACPFDT

Format

status = ACP_STD$READBLK (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_ENDOFFILE	End of file reached.
SS$_FILNOTACC	File not accessed on channel.
SS$_NOPRIV	Process has insufficient privileges.
SS$_ILLIOFUNC	Illegal I/O function.

SS$_ILLBLKNUM	Illegal block number.
SS$_NORMAL	Normal, successful completion.
SS$_INSFWSL	Insufficient working set limit.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$READBLK as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$READBLK (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$READBLK.
- ACP$READBLK returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$READBLK returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$WRITEBLK

Processes a write block ACP function.

Module

SYSACPFDT

Format

status = ACP_STD$WRITEBLK (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_BADPARAM	Record size is too small for magtape function processing.
SS$_ENDOFFILE	End of file reached.
SS$_FILNOTACC	File not accessed on channel.
SS$_NOPRIV	Process has insufficient privileges.

SS$_ILLIOFUNC	Illegal I/O function.
SS$_ILLBLKNUM	Illegal block number.
SS$_INSFMEM	Insufficient memory to perform erase function.
SS$_INSFSPTS	Insufficient system page table entries to perform erase function.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Normal, successful completion.
SS$_WRITLCK	Device software is write locked.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$WRITEBLK as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine ACP$WRITEBLK (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$WRITEBLK.
- ACP$WRITEBLK returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$WRITEBLK returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

COM_STD$DELATTNAST

Delivers all attention ASTs linked in the specified list.

Module

COMDRVSUB

Format

COM_STD$DELATTNAST (acb_lh, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	address	input	reference	required
ucb	UCB	input	reference	required

ast_lh
Listhead of AST control blocks

ucb
Unit control block.

Context

COM_STD$DELATTNAST executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

COM_STD$DELATTNAST removes all AST control blocks (ACBs) from the specified list. Using each ACB as a fork block, it schedules a fork process at IPL$_QUEUEAST to queue the AST to its target process. COM_STD$DELATTNAST dequeues each ACB from the head of the list, thus removing them in the reverse order of their declaration by COM_STD$SETATTNAST. Note that in certain circumstances attention ASTs can be delivered to a user process before the delivery of I/O completion ASTs previously posted by the driver.

Macro

$DELATTNAST [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELATTNAST. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, $DELATTNAST simulates a JSB to COM$DELATTNAST. It calls COM_STD$DELATTNAST using the current contents of R4 and R5 as the **listhead** and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- COM_STD$DELATTNAST replaces COM$DELATTNAST (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike COM$DELATTNAST, COM_STD$DELATTNAST does not preserve the contents of R0 and R1.

COM_STD$DELATTNASTP

Delivers all attention ASTs linked in the specified list for a given process.

Module

COMDRVSUB

Format

COM_STD$DELATTNASTP (acb_lh, ucb, ipid)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	listhead	input	reference	required
ucb	UCB	input	reference	required
ipid	integer	input	value	required

acb_lh
Listhead of AST control blocks

ucb
Unit control block.

ipid
Internal process ID (IPID) for the target process.

Context

COM_STD$DELATTNASTP executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

For Digital internal use only.

Macro

$DELATTNASTP [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELATTNASTP. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, $DELATTNASTP simulates a JSB to COM$DELATTNASTP. It calls COM_STD$DELATTNASTP using the current contents of R4, R5 and R6 as the **listhead**, **ucb**, and **ipid** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- COM_STD$DELATTNASTP replaces COM$DELATTNASTP (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike COM$DELATTNASTP, COM_STD$DELATTNASTP does not preserve the contents of R0 and R1.

COM_STD$DELCTRLAST

Delivers all control ASTs, linked in the specified list, that match a given condition.

Module

COMDRVSUB

Format

COM_STD$DELCTRLAST (acb_lh, ucb, matchchar, inclchar_p)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	listhead	input	reference	required
ucb	UCB	input	reference	required
matchchar	integer	input	value	required
inclchar_p	pointer	output	value	required

acb_lh
Listhead of AST control blocks

ucb
Unit control block.

matchchar
Match character.

inclchar_p
Address in which COM_STD$DELCTRLAST writes the character to include in the data stream, or NULL.

Context

COM_STD$DELCTRLAST executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

For Digital internal use only.

Macro

$DELCTRLAST [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELCTRLAST. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, $DELCTRLAST simulates a JSB to COM$DELCTRLAST. It calls COM_STD$DELCTRLAST using the current contents of R4, R5, and R3 as the **listhead**, **ucb**, and **matchchar** arguments, respectively. When COM$DELCTRLAST returns, it moves the include character itno R3. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- COM_STD$DELCTRLAST replaces COM$DELCTRLAST (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike COM$DELCTRLAST, COM_STD$DELCTRLAST does not preserve the contents of R0 and R1.

COM_STD$DELCTRLASTP

Delivers all control ASTs, linked in the specified list, that match a given condition.

Module

COMDRVSUB

Format

COM_STD$DELCTRLASTP (acb_lh, ucb, ipid, matchchar, inclchar_p)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	listhead	input	reference	required
ucb	UCB	input	reference	required
ipid	integer	input	value	required
matchchar	integer	input	value	required
inclchar_p	pointer	input	value	required

acb_lh
Listhead of AST control blocks

ucb
Unit control block.

ipid
Internal process ID (IPID) for the target process.

matchchar
Match character.

inclchar_p
Address in which COM_STD$DELCTRLAST writes the character to include in the data stream, or NULL.

Context

COM_STD$DELCTRLASTP executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

For Digital internal use only.

Macro

$DELCTRLASTP [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELCTRLASTP. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, $DELCTRLASTP simulates a JSB to COM$DELCTRLASTP. It calls COM_STD$DELCTRLASTP using the current contents of R4, R5, R6, and R3 as the **listhead**, **ucb**, **ipid**, and **matchchar** arguments, respectively. When COM$DELCTRLASTP returns, it moves the include character itno R3. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- COM_STD$DELCTRLASTP replaces COM$DELCTRLASTP (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike COM$DELCTRLASTP, COM_STD$DELCTRLASTP does not preserve the contents of R0 and R1.

COM_STD$DRVDEALMEM

Deallocates system dynamic memory.

Module

MEMORYALC_MIN or MEMORYALC_MON

Format

COM_STD$DRVDEALMEM (block)

Arguments

Argument	Type	Access	Mechanism	Status
ptr	structure	input	reference	required

ptr
Block to be deallocated. The block must be a standard OpenVMS data structure (in which offset FKB$W_SIZE contains its size). The block size must be at least FKB$K_LENGTH (24 bytes). (The FKB$ symbols are defined by the $FKBDEF macro in SYS$LIBRARY:LIB.MLB.)

Context

A driver can call COM_STD$DRVDEALMEM from any IPL. COM_STD$DRVDEALMEM executes at the caller's IPL and returns control at that IPL. The caller retains any spin locks it held at the time of the call.

Description

COM_STD$DRVDEALMEM transfers control to EXE$DEANONPAGED to deallocate the buffer specified by the **block** parameter. If COM_STD$DRVDEALMEM cannot deallocate memory at the caller's IPL, it transforms the block being deallocated into a fork block and queues the block in the fork queue. The code that executes in the fork process then jumps to EXE$DEANONPAGED.

If the buffer to be deallocated is less than FKB$C_LENGTH in size, or its address is not aligned on a 16-byte boundary, COM_STD$DRVDEALMEM issues a BADDALRQSZ bugcheck.

Macro

$DRVDEALMEM [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DRVDEALMEM. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, $DRVDEALMEM simulates a JSB to COM$DRVDEALMEM. It calls COM_STD$DRVDEALMEM using the current contents of R0 as the **ptr** argument. Unless you specify **save_r0r1=NO**, the macro preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- COM_STD$DRVDEALMEM replaces COM$DRVDEALMEM (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike COM$DRVDEALMEM, COM_STD$DRVDEALMEM does not preserve the contents of R0 and R1.

COM_STD$FLUSHATTNS

Removes specified ASTs from an attention AST list.

Module

COMDRVSUB

Format

status = COM_STD$FLUSHATTNS (pcb, ucb, chan ,acb_lh)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
chan	integer	input	value	required
acb_lh	listhead	input	reference	required

pcb

Process control block. COM_STD$FLUSHATTNS reads the following PCB fields:

Field	Contents
PCB$L_PID	Process ID
PCB$L_ASTCNT	ASTs remaining in quota

COM_STD$FLUSHATTNS increments PCB$L_ASTCNT once for each AST control block (ACB) it flushes.

ucb

Unit control block. COM_STD$FLUSHATTNS reads UCB$L_DLCK to obtain the address of the device lock.

chan

Number of the assigned I/O channel.

acb_lh

Listhead of ACBs.

Return Values

SS$_NORMAL	Normal, successful completion

Context

COM_STD$FLUSHATTNS raises IPL to device IPL, acquiring the corresponding device lock. Before returning control to its caller at the caller's IPL, COM_STD$FLUSHATTNS releases the device lock. The caller retains any spin locks it held at the time of the call.

Description

A driver's cancel-I/O routine calls COM_STD$FLUSHATTNS to flush an attention AST list. A driver FDT routine calls COM_STD$FLUSHATTNS to service a $QIO request that specifies a set-attention-AST function and a value of 0 in the **p1** argument (IRP$L_QIO_P1).

COM_STD$FLUSHATTNS locates all ACBs blocks whose channel number and PID match those supplied as input to the routine. It removes them from the specified list, deallocates them, and returns control to its caller.

Macro

$FLUSHATTNS

In a Step 2 OpenVMS AXP driver, $FLUSHATTNS simulates a JSB to COM$FLUSHATTNS. It calls COM_STD$FLUSHATTNS using the current contents of R4, R5, R6, and R7 as the **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

Notes for Converting Step 1 Drivers

- COM_STD$FLUSHATTNS replaces COM$FLUSHATTNS (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

COM_STD$FLUSHCTRLS

Removes specified ASTs from a control AST list.

Module

COMDRVSUB

Format

status = COM_STD$FLUSHCTRLS (pcb, ucb, chan ,acb_lh, mask_p)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
chan	integer	input	value	required
acb_lh	listhead	input	reference	required
mask_p	mask_ longword	input	reference	required

pcb
Process control block. COM_STD$FLUSHCTRLS reads the following PCB fields:

Field	Contents
PCB$L_PID	Process ID
PCB$L_ASTCNT	ASTs remaining in quota

COM_STD$FLUSHCTRLS increments PCB$L_ASTCNT once for each control AST control block (TAST) it flushes.

ucb
Unit control block. COM_STD$FLUSHCTRLS reads UCB$L_DLCK to obtain the address of the device lock.

chan
Number of the assigned I/O channel.

acb_lh
Listhead of ACBs.

mask_p
Summary mask of active control characters. COM_STD$FLUSHCTRLS updates this mask.

Return Values

SS$_NORMAL	Normal, successful completion

Context

COM_STD$FLUSHCTRLS raises IPL to device IPL, acquiring the corresponding device lock. Before returning control to its caller at the caller's IPL, COM_STD$FLUSHCTRLS releases the device lock. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

$FLUSHCTRLS

In a Step 2 OpenVMS AXP driver, $FLUSHCTRLS simulates a JSB to COM$FLUSHCTRLS. It calls COM_STD$FLUSHCTRLS using the current contents of R2, R4, R5, R6, and R7 as the **mask**, **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

Notes for Converting Step 1 Drivers

- COM_STD$FLUSHCTRLS replaces COM$FLUSHCTRLS (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

COM_STD$POST, COM_STD$POST_NOCNT

Initiate device-independent postprocessing of an I/O request independent of the status of the device unit.

Module

COMDRVSUB

Format

COM_STD$POST (irp, ucb)

COM_STD$POST_NOCNT (irp)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request block. The following IRP fields are input to I/O postprocessing.

Field	Contents
IRP$L_MEDIA	Data to be copied to the I/O status block
IRP$L_MEDIA+4	Data to be copied to the I/O status block

ucb
Unit control block (COM_STD$POST only). COM_STD$POST increments the unit operation count (UCB$L_OPCNT).

Context

Drivers call COM_STD$POST at or above fork IPL. Drivers call COM_STD$POST_NOCNT at or above IPL$_ASTDEL. These routines execute at their caller's IPL and return control at that IPL. The caller retains any spin locks it held at the time of the call.

Description

A driver fork process calls COM_STD$POST or COM_STD$POST_NOCNT after it has completed device-dependent I/O processing for an I/O request initiated by EXE_STD$ALTQUEPKT. Because COM_STD$POST_NOCNT, unlike COM_STD$POST, does not increment the unit's operations count (UCB$L_OPCNT), a driver uses COM_STD$POST_NOCNT to initiate completion processing for an I/O request when the associated UCB is not available.

COM_STD$POST and COM_STD$POST_NOCNT insert the IRP into the systemwide I/O postprocessing queue, request an IPL$_IOPOST software interrupt, and return control to the caller. Unlike IOC_STD$REQCOM, these routines do not attempt to dequeue any IRP waiting for the device or change the busy status of the device.

Macro

```
$POST [save_r1]
$POST_NOCNT [save_r1]
```

where:

save_r1 indicates that the macro should preserve register R1 across the call to COM_STD$POST or COM_STD$POST_NOCNT. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, $POST simulates a JSB to COM$POST. It calls COM_STD$POST using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. $POST_NOCNT simulates a JSB to COM$POST_NOCNT. It calls COM_STD$POST_NOCNT using the current contents of R3 as the **irp** argument. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call.

Notes for Converting Step 1 Drivers

- COM_STD$POST replaces COM$POST (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers); COM_STD$POST_NOCNT replaces COM$POST_NOCNT. Unlike the Step 1 routines, the Step 2 routines do not preserve R1 across the call.

COM_STD$SETATTNAST

Enables or disables attention ASTs.

Module

COMDRVSUB

Format

status = COM_STD$SETATTNAST (irp, pcb, ucb, ccb, acb_lh)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
acb_lh	listhead	input	reference	required

irp

I/O request packet for the current I/O request.

COM_STD$SETATTNAST reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the address of the AST routine, or zero to flush the AST queue.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the AST parameter.
IRP$L_QIO_P3	$QIO system service **p3** argument, containing the access mode of the AST request.
IRP$L_CHAN	I/O request channel index number.

pcb

Process control block of the current process.

COM_STD$SETATTNAST reads the following PCB fields:

Field	Contents
PCB$L_ASTCNT	Number of ASTs remaining in process quota
PCB$L_PID	Process ID

COM_STD$SETATTNAST decrements PCB$L_ASTCNT if it successfully queues the AST.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

COM_STD$SETATTNAST reads UCB$L_DLCK.

ccb
Channel control block that describes the process-I/O channel

acb_lh
Address of listhead of AST control blocks.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.
SS$_NORMAL	Normal, successful completion

Status in FDT_CONTEXT

SS$_EXQUOTA	Process AST quota exceeded.
SS$_INSFMEM	No memory available to allocate the expanded ACB.

Context

The FDT support routine COM_STD$SETATTNAST must be called from code executing at IPL$_ASTDEL. COM_STD$SETATTNAST raises IPL and acquires the corresponding device lock, to insert the AST into the AST queue. It returns control to its caller at IPL$_ASTDEL.

Description

A driver FDT routine calls COM_STD$SETATTNAST to service a $QIO request that specifies a set-attention-AST function.

If the **p1** argument of the request contains a zero, COM_STD$SETATTNAST transfers control to COM_STD$FLUSHATTNS, which disables all ASTs indicated by the PID and I/O channel number (IRP$L_CHAN). COM_STD$FLUSHATTNS searches through the AST control block (ACB) list, extracts each identified ACB, deallocates it, and returns SS$_NORMAL status in R0 to COM_STD$SETATTNAST. COM_STD$SETATTNAST returns this status to its caller.

If the **p1** argument of the request contains the address of an AST routine, COM_STD$SETATTNAST decrements PCB$L_ASTCNT and allocates an expanded AST control block (ACB) that contains the following information:

- Spin lock index SPL$C_QUEUEAST
- Address of the AST routine (as specified in **p1**)
- AST parameter (as specified in **p2**)
- Access mode (the maximum, or least privileged, access mode between the access mode specified in **p3** and the current process's access mode). Bit ACB$V_QUOTA is set in this value to indicate a that the AST was requested by a process, not by the system.)

- Number of the assigned I/O channel
- PID of the requesting process

COM_STD$SETATTNAST links the ACB to the start of the specified linked list of ACBs located in a UCB extension area. COM$DELATTNAST can later use the expanded ACB to fork to IPL$_QUEUEAST, at which IPL it reformats the block into a standard ACB.

If the process exceeds its AST quota, or if there is no memory available to allocate the expanded ACB, COM_STD$SETATTNAST restores PCB$L_ASTCNT to its original value and calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, COM_STD$SETATTNAST returns to its caller with this status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

The caller of COM_STD$SETATTNAST must examine the status in R0:

- If the status is SS$_NORMAL, the attention AST has been enabled (or the AST has been flushed), as requested.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the operation to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS.

Macro

$SETATTNAST

In a Step 2 OpenVMS AXP driver, $SETATTNAST simulates a JSB to COM$SETATTNAST. It calls COM_STD$SETATTNAST using the current contents of R3, R4, R5, R6, and R7, as the **irp**, **pcb**, **ucb**, **ccb**, and **acb_lh** arguments, respectively. It returns status in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- COM_STD$SETATTNAST replaces COM$SETATTNAST (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). COM$SETATTNAST returns to its caller only upon success; COM_STD$SETATTNAST returns to its caller whether it has been successful or not. It returns SS$_NORMAL or SS$_FDT_COMPL status in R0. When it returns SS$_FDT_COMPL status, the FDT_CONTEXT structure contains additional status (SS$_EXQUOTA or SS$_INSFMEM) to explain why the request has been aborted.
- COM$SETATTNAST preserves the addresses of the IRP and UCB in R3 and R5 across the call. COM_STD$SETATTNAST does not.

COM_STD$SETCTRLAST

Enables or disables control ASTs.

Module

COMDRVSUB

Format

status = COM_STD$SETCTRLAST (irp, pcb, ucb, acb_lh, mask, tast_p)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
acb_lh	listhead	input	reference	required
mask	mask_ longword	input	value	required
tast_p	TAST	output	value	required

irp

I/O request packet for the current I/O request.

COM_STD$SETCTRLAST reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the address of the AST routine to call when an out-of-band character is typed, or zero to flush the queue.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the address of the short-form terminator mask, indicating which out-of-band characters precipitate AST delivery. This address is passed as an AST parameter when the AST is delivered.
IRP$L_QIO_P3	$QIO system service **p3** argument, containing the access mode of the AST request.
IRP$L_CHAN	I/O request channel index number

pcb

Process control block of the current process.

COM_STD$SETCTRLAST reads the following PCB fields:

Field	Contents
PCB$L_ASTCNT	Number of ASTs remaining in process quota
PCB$L_PID	Process ID

COM_STD$SETCTRLAST decrements PCB$L_ASTCNT if it successfully queues the AST.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

COM_STD$SETCTRLAST reads UCB$L_DLCK.

acb_lh
Address of listhead of AST control blocks.

mask
Summary mask of active control characters. COM_STD$SETCTRLAST updates the summary mask to be the inclusive-OR of all masks in the control AST list.

tast_p
Address of the control AST block (TAST), returned as output from COM_STD$SETCTRLAST.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.
SS$_NORMAL	Normal, successful completion

Status in FDT_CONTEXT

SS$_ACCVIO	Specified **mask** is not addressable.
SS$_EXQUOTA	Process AST quota exceeded.
SS$_INSFMEM	No memory available to allocate the expanded ACB.

Context

The FDT support routine COM_STD$SETCTRLAST must be called from code executing at IPL$_ASTDEL. COM_STD$SETCTRLAST raises IPL and acquires the corresponding device lock, to insert the AST into the AST queue. It returns control to its caller at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$SETCTRLAST

In a Step 2 OpenVMS AXP driver, $SETCTRLAST simulates a JSB to COM$SETCTRLAST. It calls COM_STD$SETCTRLAST using the current contents of R3, R4, R5, R7, and R2, as the **irp**, **pcb**, **ucb**, **acb_lh**, and **mask** arguments, respectively. It returns the TAST block in R2. It returns status in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- COM_STD$SETCTRLAST replaces COM$SETCTRLAST (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). The order in which formal parameters are passed to COM_STD$SETCTRLAST differs from the order in which they are provided in registers to the Step 1 OpenVMS routine COM$SETCTRLAST.
- COM_STD$SETCTRLAST does not provide the address of the TAST block as output in R2.
- COM$SETCTRLAST returns to its caller only upon success; COM_STD$SETCTRLAST returns to its caller whether it has been successful or not. It returns SS$_NORMAL or SS$_FDT_COMPL status in R0. When it returns SS$_FDT_COMPL status, the FDT_CONTEXT structure contains additional status (SS$_EXQUOTA or SS$_INSFMEM) to explain why the request has been aborted.

ERL_STD$ALLOCEMB

Allocates an error log message buffer and initializes its header.

Module

ERRORLOG

Format

status = ERL_STD$ALLOCEMB (size, embdv_p)

Arguments

Argument	Type	Access	Mechanism	Status
size	integer	input	value	required
embdv_p	pointer	output	reference	required

size
Size of requested error message buffer in bytes.

embdv_p
Address of a pointer in which ERL_STD$ALLOCEMB writes the address of the error message buffer (EMBDV).

Return Values

status	Low bit set indicates success, low bit clear indicates failure

Context

A driver can call ERL_STD$ALLOCEMB from any IPL. ERL_STD$ALLOCEMB raises IPL to IPL$_EMB and obtains the corresponding spin lock to allocate the error message buffer. It returns control to its caller at its caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

$ALLOCEMB

In a Step 2 OpenVMS AXP driver, $ALLOCEMB simulates a JSB to ERL$ALLOCEMB. It calls ERL_STD$ALLOCEMB using the current contents of R1 as the **size** argument. It returns status in R0, the address of the allocated EMB in R2 and copies the error log sequence number from EMB$W_DV_ERRSEQ to R1.

Notes for Converting Step 1 Drivers

- ERL_STD$ALLOCEMB replaces ERL$ALLOCEMB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike ERL$ALLOCEMB, ERL_STD$ALLOCEMB does not return the error sequence number in R1. A driver can obtain the error sequence number from the error message buffer (EMB$W_DV_ERRSEQ).

ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, ERL_STD$DEVICTMO

Allocate an error message buffer and record in it information concerning the error.

Module

ERRORLOG

Format

ERL_STD$DEVICEATTN (driver_param, ucb)

ERL_STD$DEVICERR (driver_param, ucb)

ERL_STD$DEVICTMO (driver_param, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
driver_param	undefined	input	reference	required
ucb	UCB	input	reference	required

driver_param
Parameter to be passed to the register dumping routine, usually a controller register access mailbox (CRAM).

ucb
Unit control block. These routines read the following UCB fields:

Field	Contents
UCB$L_DEVCHAR	Bit DEV$V_ELG set.
UCB$L_FUNC	Bit IO$V_INHERLOG clear.
UCB$L_IRP	Address of IRP currently being processed (ERL_STD$DEVICERR and ERL_STD$DEVICTMO only).
UCB$L_ORB	ORB address.
UCB$L_DDB	DDB address.
UCB$L_DDT	DDT address. DDT$W_ERRORBUF contains the size of the error message buffer in bytes.

They write the following UCB fields:

Field	Contents
UCB$L_ERRCNT	Incremented.
UCB$L_EMB	Address of error message buffer.
UCB$L_STS	UCB$V_ERLOGIP set.

Context

A driver calls ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, or ERL_STD$DEVICTMO at or above fork IPL, holding the corresponding fork lock in an OpenVMS multiprocessing environment.

These routines return control to the caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

ERL_STD$DEVICERR and ERL_STD$DEVICTMO log an error associated with a particular I/O request. ERL_STD$DEVICEATTN logs an error that is not associated with an I/O request. Each of these routines performs the following steps:

- Increments UCB$L_ERRCNT to record a device error. If the error-log-in-progress bit (UCB$V_ERLOGIP in UCB$L_STS) is set, the routine returns control to its caller.
- Allocates from the current error log allocation buffer an error message buffer of the length specified in the device's DDT (in argument **erlgbf** to the DDTAB macro). This allocation is performed at IPL$_EMB holding the EMB spin lock.
- Places the address of the error message buffer in UCB$L_EMB.
- Sets UCB$V_ERLOGIP in UCB$L_STS.
- Initializes the buffer with the current system time, error log sequence number, and error type code. These routines use the following error type codes:

ERL_STD$DEVICEATTN	Device attention (EMB$C_DA)
ERL_STD$DEVICERR	Device error (EMB$C_DE)
ERL_STD$DEVICTMO	Device timeout (EMB$C_DT)

- Loads fields from the UCB, the IRP, and the DDB into the buffer, including the following:

UCB$B_DEVCLASS	Device class
UCB$B_DEVTYPE	Device type
IRP$L_PID	Process ID of the process originating the I/O request (ERL_STD$DEVICERR or ERL_STD$DEVICTMO)
IRP$L_BOFF	Transfer parameter (ERL_STD$DEVICERR and ERL_STD$DEVICTMO)
IRP$L_BCNT	Transfer parameter (ERL_STD$DEVICERR and ERL_STD$DEVICTMO)
IRP$L_MEDIA	Disk address
UCB$W_UNIT	Unit number
UCB$L_ERRCNT	Count of device errors
UCB$L_OPCNT	Count of completed operations
ORB$L_OWNER	UIC of volume owner

UCB$L_DEVCHAR	Device characteristics
IRP$L_FUNC	I/O function value (ERL_STD$DEVICERR and ERL_STD$DEVICTMO)
DDB$T_NAME	Device name (concatenated with cluster node name if appropriate)

- Loads into R0 the address of the location in the buffer in which the contents of the device registers are to be stored.
- Calls the driver's register dumping routine, the address of which is specified in the **regdmp** argument to the DDTAB macro.

Macro

```
$DEVICEATTN [save_r0r1]
$DEVICERR [save_r0r1]
$DEVICTMO [save_r0r1]
```

where:

save_r0r1 indicates that the macros must preserve the contents of R0 and R1 across the call to ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, or ERL_STD$DEVICTMO. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, the $DEVICEATTN, $DEVICERR, and $DEVICTMO macros simulate JSBs to ERL$DEVICEATTN, ERL$DEVICERR, and ERL$DEVICTMO, respectively. Each macro calls the corresponding routine using the current contents of R4 and R5 as the **driverpar** and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, and ERL_STD$DEVICTMO replace ERL$DEVICEATTN, ERL$DEVICERR, and ERL$DEVICTMO (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike the Step 1 routines, the Step 2 routines do not preserve the contents of R0 and R1.
- Because the UCB$L_MEDIA field has been removed from the UCB local disk extension, these routines write the disk address into the EMB from IRP$L_MEDIA.
- Because the UCB$B_SLAVE field has been removed from the UCB local disk extension, these routines do not write that field.
- OpenVMS AXP device drivers consequently do not need to define the local disk UCB extension or local tape UCB extension to use these error logging routines.
- **driver_param** is considered required input to these routines.

ERL_STD$RELEASEMB

Releases an error message buffer to the error logging process.

Module

ERRORLOG

Format

ERL_STD$RELEASEMB (embdv)

Arguments

Argument	Type	Access	Mechanism	Status
embdv	EMBDV	input	reference	required

embdv
Error message buffer to be released.

Context

A driver can call ERL_STD$RELEASEMB from any IPL. ERL_STD$RELEASEMB raises IPL to IPL$_EMB and obtains the corresponding spin lock to release the error message buffer. It returns control to its caller at its caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

$RELEASEMB

In a Step 2 OpenVMS AXP driver, $RELEASEMB simulates a JSB to ERL$RELEASEMB. It calls ERL_STD$RELEASEMB using the current contents of R2 as the **buff** argument.

Notes for Converting Step 1 Drivers

- ERL_STD$RELEASEMB replaces ERL$RELEASEMB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

EXE_STD$ABORTIO

Completes the servicing of an I/O request without returning status to the I/O status block specified in the request.

Module

SYSQIOREQ

Format

status = EXE_STD$ABORTIO (irp, pcb, ucb, qio_sts)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
qio_sts	integer	input	value	required

irp

I/O request packet. EXE_STD$ABORTIO copies the **qio_sts** parameter to IRP$L_IOST1 and clears IRP$PS_FDT_CONTEXT. The caller of EXE_STD$ABORTIO should not access the IRP after the routine returns SS$_FDT_COMPL status.

pcb

PCB of current process

ucb

Unit control block

qio_sts

Final status to be returned by the $QIO system service to its caller. EXE_STD$ABORTIO places this status in FDT_CONTEXT$L_QIO_STATUS. If you intend to access the FDT context structure after EXE_STD$ABORTIO returns, you must obtain its address from IRP$PS_FDT_CONTEXT and store it before making the call.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

Contents of **qio_sts** argument

Context

EXE_STD$ABORTIO executes at its caller's IPL and raises to fork IPL, acquiring the associated fork lock in a multiprocessing environment. As a result, its caller cannot be executing above fork IPL. A driver usually transfers control to EXE_STD$ABORTIO at IPL$_ASTDEL.

EXE_STD$ABORTIO returns to its caller at the caller's IPL.

Description

The FDT completion routine EXE_STD$ABORTIO terminates the servicing of an I/O request without returning status to the I/O status block specified in the original call to the $QIO system service.

EXE_STD$ABORTIO performs the following actions:

1. Examines the **qio_sts** argument. If the argument contains SS$_FDT_COMPL, EXE_STD$ABORTIO returns to its caller. This check prevents an I/O request from being aborted more than once.
2. Places the status to be returned to the caller of the $QIO system service in IRP$L_IOST1 and in the FDT_CONTEXT structure.
3. Clears the pointer to the FDT_CONTEXT structure in IRP$PS_FDT_CONTEXT.
4. Requests the fork lock, raising IPL to fork IPL, to perform the following tasks:
   - Clear IRP$L_IOSB so that no status is returned by I/O postprocessing
   - Clear ACB$V_QUOTA in IRP$B_RMOD to prevent the delivery of any AST to the process specified in the I/O request
   - Update the count of available AST entries at PCB$L_ASTCNT, if necessary
   - Insert the IRP in the local processor's I/O postprocessing queue. If the queue is empty, request a software interrupt from the local processor at IPL$_IOPOST.
5. Releases the fork lock, restoring the caller's IPL. The pending IPL$_IOPOST interrupt causes I/O postprocessing to occur before the remaining instructions in EXE_STD$ABORTIO are executed.

When all I/O postprocessing has been completed, EXE_STD$ABORTIO regains control and returns SS$_FDT_COMPL status to its caller.

Any ASTs specified when the I/O request was issued will not be delivered, and any event flags requested will not be set.

Macro

$ABORTIO [do_ret=YES]

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In a Step 2 OpenVMS AXP driver, the $ABORTIO macro simulates the JMP to EXE$ABORTIO in the FDT routine of a Step 1 driver. It initializes the **irp**, **pcb**, **ucb**, and **qio_sts** parameters from the contents of R3, R4, R5, and R0, respectively, and calls EXE_STD$ABORTIO. When EXE_STD$ABORTIO returns control to the code generated by a default invocation of $ABORTIO, a RET instruction returns control to the caller of $ABORTIO's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- The order in which formal parameters are passed to EXE_STD$ABORTIO differs from the order in which they are provided in registers to the Step 1 OpenVMS routine EXE$ABORTIO.
- The contents of R0 (final $QIO system service status in a call to EXE$ABORTIO in a Step 1 driver) are destroyed across the call to EXE_STD$ABORTIO. This is especially important if you use the $ABORTIO macro on OpenVMS AXP systems and expect R0 to retain its value afterwards.
- Unlike EXE$ABORTIO, EXE_STD$ABORTIO does not lower IPL to 0 before exiting. EXE_STD$ABORTIO returns to its caller at the caller's IPL.
- EXE$ABORTIO returns control to the system service dispatcher, passing it the final $QIO system service status in R0. EXE_STD$ABORTIO returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

<

EXE_STD$ALLOCBUF, EXE_STD$ALLOCIRP

Allocates a buffer from nonpaged pool for a buffered-I/O operation.

Module

MEMORYALC

Format

status = EXE_STD$ALLOCBUF (reqsize, blocksize, blockptr)

status = EXE_STD$ALLOCIRP (blocksize, blockptr)

Arguments

Argument	Type	Access	Mechanism	Status
reqsize	integer	input	value	required
alosize_p	pointer	output	value	required
bufptr_p	pointer	output	value	required

reqsize
Size of requested buffer in bytes (EXE_STD$ALLOCBUF only). This value should include the 12 bytes required to store header information.

alosize_p
Location in which EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP write the size of the requested buffer in bytes.

bufptr_p
Location in which EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP write the address of allocated buffer. The following fields are initialized in the buffer:

Field	Contents
IRP$W_SIZE (in allocated buffer)	Size of requested buffer in bytes (for EXE_STD$ALLOCBUF), IRP$C_LENGTH (for EXE_STD$ALLOCIRP).
IRP$B_TYPE (in allocated buffer)	DYN$C_BUFIO (for EXE_STD$ALLOCBUF), DYN$C_IRP (for EXE_STD$ALLOCIRP).

Return Values

SS$_NORMAL	Normal, successful completion.
SS$_INSFMEM	Insufficient memory to satisfy request.

Context

EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP set IPL to IPL$_ASTDEL. As a result they cannot be called by code executing above IPL$_ASTDEL. They return control to the caller at IPL$_ASTDEL.

Description

EXE_STD$ALLOCBUF attempts to allocate a buffer of the requested size from nonpaged pool; EXE_STD$ALLOCIRP attempts to allocate an IRP from nonpaged pool.

If sufficient memory is not available, EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP examine the PCB (CTL$GL_PCB) to determine whether the process has resource wait mode enabled. If PCB$V_SSRWAIT in PCB$L_STS is clear, these routines place the process in a resource wait state until memory is released.

The caller must check and adjust process quotas (JIB$L_BYTCNT or JIB$L_BYTLM, or both) by calling EXE$DEBIT_BYTCNT or EXE$DEBIT_BYTCNT_BYTLM. (Note that you can perform this task and allocate a buffer of the requested size by using the routines EXE$DEBIT_BYTCNT_ALO and EXE$DEBIT_BYTCNT_BYTLM_ALO. These routines invoke EXE_STD$ALLOCBUF.)

The normal buffered I/O postprocessing routine (IOC_STD$REQCOM), initiated by the REQCOM macro, readjusts quotas and also deallocates the buffer.

Note that the value returned in the **alosize_p** argument and placed at IRP$W_SIZE in the allocated buffer is the size of the allocated buffer. The actual size of the buffer is determined according to the algorithms used by EXE$ALONONPAGED and the size of the lookaside list packets. The nonpaged pool deallocation routine (EXE$DEANONPAGED), called in buffered I/O postprocessing, uses similar algorithms when returning memory to nonpaged pool.

Macro

$ALLOCBUF
$ALLOCIRP

In a Step 2 OpenVMS AXP driver, $ALLOCBUF and $ALLOCIRP simulate a JSB to EXE$ALLOCBUF and EXE$ALLOCIRP, respectively. $ALLOCBUF calls EXE_STD$ALLOCBUF using the current contents of R1 as the **reqsize** argument. Both $ALLOCBUF and $ALLOCIRP return status in R0, the address of the allocated buffer in R2 and its size in R1. If a resource wait occurred, these macros return the address of the PCB in R4.

Notes for Converting Step 1 Drivers

- EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP replace EXE$ALLOCBUF and EXE$ALLOCIRP (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike the Step 1 routines, the Step 2 routines do not preserve the original contents of R4, or return the address of the PCB in R4 if a wait has occurred.

EXE_STD$ALTQUEPKT

Delivers an IRP to a driver's alternate start-I/O routine without regard for the status of the device.

Module

SYSQIOREQ

Format

EXE_STD$ALTQUEPKT (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet.

ucb
Unit control block.

EXE_STD$ALTQUEPKT reads the following UCB fields:

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_DDT	Address of unit's DDT. EXE_STD$ALTQUEPKT reads DDB$PS_ALTSTART to obtain the procedure value of the driver's alternate start-I/O routine.
UCB$L_ALTIOWQ	Address of the alternate start-I/O wait queue listhead.

Context

A driver FDT routine typically calls EXE_STD$ALTQUEPKT at IPL$_ASTDEL. EXE_STD$ALTQUEPKT raises to fork IPL (acquiring the associated fork lock) before calling the driver's alternate start-I/O routine. When the alternate start-I/O routine returns control to it, EXE_STD$ALTQUEPKT returns control to its caller at the caller's IPL (having released its acquisition of the fork lock).

Description

EXE_STD$ALTQUEPKT calls the driver's alternate start-I/O routine. It does not test whether the unit is busy before making the call.

Macro

$ALTQUEPKT

In a Step 2 OpenVMS AXP driver, the $ALTQUEPKT macro simulates the JSB to EXE$ALTQUEPKT in the FDT routine of a Step 1 driver. $ALTQUEPKT calls EXE_STD$ALTQUEPKT, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- EXE_STD$ALTQUEPKT replaces EXE$ALTQUEPKT (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

EXE_STD$CARRIAGE

Interprets the carriage control specifier in IRP$B_CARCON and converts it to a generic prefix/suffix format.

Module

SYSQIOFDT

Format

EXE_STD$CARRIAGE (irp)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required

irp
I/O request packet.

Context

A driver FDT routine calls EXE_STD$CARRIAGE at IPL$_ASTDEL. EXE_STD$CARRIAGE returns control to the driver at that IPL.

Description

For Digital internal use only.

Macro

$CARRIAGE

In a Step 2 OpenVMS AXP driver, the $CARRIAGE macro simulates the JSB to EXE$CARRIAGE in the FDT routine of a Step 1 driver. $CARRIAGE calls EXE_STD$CARRIAGE, using the current contents of R3 as the **irp** arguments.

Notes for Converting Step 1 Drivers

- EXE_STD$CARRIAGE replaces EXE$CARRIAGE (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

EXE_STD$CHK*xxx*ACCES

Checks logical (EXE_STD$CHKLOGACCES), physical (EXE_STD$CHKPHYACCES), read (EXE_STD$CHKRDACCES), write (EXE_STD$CHKWRTACCES), execute (EXE_STD$CHKEXEACCES), create (EXE_STD$CHKCREACCES), or delete (EXE_STD$CHKDELACCES) I/O function access, based on the specified protection information.

Module

EXSUBROUT

Format

status = EXE_STD$CHKCREACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKDELACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKEXEACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKLOGACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKPHYACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKRDACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKWRTACCES (arb, orb, pcb, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
arb	ARB	input	reference	required
orb	ORB	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required

arb
Agent rights block.

orb
Object rights block.

pcb
Process control block of accessor.

ucb
Unit control block of accessed object.

Return Values

SS$_NORMAL	Specified access allowed.
SS$_NOPRIV	Specified access denied.

Context

A driver FDT routine calls EXE_STD$CHKPHYACCES, EXE_STD$CHKLOGACCES, EXE_STD$CHKWRTACCES, EXE_STD$CHKRDACCES, EXE_STD$CHKCREACCES, EXE_STD$CHKEXEACCES, and EXE_STD$CHKDELACCES, at IPL$_ASTDEL. These routines return control to the driver at that IPL.

Description

For Digital internal use only.

Macro

```
$CHKCREACCES [save_r1]
$CHKDELACCES [save_r1]
$CHKEXEACCES [save_r1]
$CHKLOGACCES [save_r1]
$CHKPHYACCES [save_r1]
$CHKRDACCES [save_r1]
$CHKWRTACCES [save_r1]
```

where:

save_r1 indicates that the macro must preserve the contents of R1 across the call to EXE_STD$CHKPHYACCES, EXE_STD$CHKLOGACCES, EXE_STD$CHKWRTACCES, EXE_STD$CHKEXEACCES, EXE_STD$CHKCREACCES, EXE_STD$CHKDELACCES or EXE_STD$CHKRDACCES. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, the $CHKCREACCES, $CHKDELACCES, $CHKEXEACCES, $CHKLOGACCES, $CHKPHYACCES, $CHKWRTACCES, and $CHKRDACCES, macros simulate the JSB to EXE$CHKCREACCES, EXE$CHKDELACCES, EXE$CHKEXEACCES, EXE$CHKPHYACCES, EXE$CHKLOGACCES, EXE$CHKWRTACCES, or EXE$CHKRDACCES in a Step 1 driver. Each macro calls the corresponding access-checking routine, using the current contents of R0, R1, R4, and R5 as the **arb**, **orb**, **pcb**, and **ucb** arguments. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call. All macros return status in R0.

Notes for Converting Step 1 Drivers

- The OpenVMS AXP I/O function access checking routines replace their OpenVMS AXP Step 1 and OpenVMS VAX counterparts, but do not does not preserve R1 across a call.

EXE_STD$FINISHIO

Completes the servicing of an I/O request and returns status to the I/O status block specified in the original call to the $QIO system service.

Module

SYSQIOREQ

Format

status = EXE_STD$FINISHIO (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet. EXE_STD$FINISHIO clears IRP$PS_FDT_CONTEXT. The caller of EXE_STD$FINISHIO should not access the IRP after the routine returns SS$_FDT_COMPL status.

ucb
Unit control block. EXE_STD$FINISHIO increments UCB$L_OPCNT.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$FINISHIO executes at its caller's IPL and raises to fork IPL, acquiring the associated fork lock in a multiprocessing environment. As a result, its caller cannot be executing above fork IPL. A driver usually transfers control to EXE_STD$FINISHIO at IPL$_ASTDEL.

EXE_STD$FINISHIO returns to its caller at the caller's IPL.

Description

The FDT completion routine EXE_STD$FINISHIO completes the servicing of an I/O request and returns status to the I/O status block specified in the original call to the $QIO system service. It performs the following actions:

1. Clears the pointer to the FDT context structure in IRP$PS_FDT_CONTEXT.
2. Requests the fork lock, raising IPL to fork IPL, to perform the following tasks:
   - Increase the number of I/O operations completed on the current device in the operation count field of the UCB (UCB$L_OPCNT). This task is performed at fork IPL, holding the associated fork lock in a multiprocessing environment.
   - Insert the IRP in the local processor's I/O postprocessing queue. If the queue is empty, request a software interrupt from the local processor at IPL$_IOPOST.
3. Releases the fork lock, restoring the caller's IPL. The pending IPL$_IOPOST interrupt causes I/O postprocessing to occur before the remaining instructions in EXE_STD$FINISHIO are executed.

When all I/O postprocessing has been completed, EXE_STD$FINISHIO regains control and returns SS$_FDT_COMPL status to its caller, passing SS$_NORMAL as the final $QIO completion status in the FDT_CONTEXT structure.

The image that requested the I/O operation receives SS$_NORMAL status, indicating that the I/O request has completed without device-independent error.

Macro

```
$FINISHIO [do_ret=YES]
$FINISHIOC [do_ret=YES]
```

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In a Step 2 OpenVMS AXP driver, the $FINISHIO macro simulates the JMP to EXE$FINISHIO in the FDT routine of a Step 1 driver. The $FINISHIOC macro simulates the JMP to EXE$FINISHIOC. The former macro moves the current contents of R0 and R1 into IRP$L_IOST1 and IRP$L_IOST2, respectively; the latter initializes IRP$L_IOST1 from R0 and clears IRP$L_IOST2. Both macros initialize the **irp** and **ucb** parameters from the contents of R3 and R5, respectively before calling EXE_STD$FINISHIO. When EXE_STD$FINISHIO returns control to the code generated by a default invocation of $FINISHIO or $FINISHIOC, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- A Step 1 OpenVMS AXP driver supplies the first and second longwords of device-specific status (in R0 and R1) as input to EXE$FINISHIO. EXE$FINISHIO writes these longwords to IRP$L_IOST1 and IRP$L_IOST2, respectively, from which I/O postprocessing transfers their values to the I/O status block specified in the original $QIO call. These status longwords are not input parameters to EXE_STD$FINISHIO. Rather, a Step 2 OpenVMS AXP driver's FDT routine must fill in IRP$L_IOST1 and IRP$L_IOST2 before calling EXE_STD$FINISHIO.

  Because the OpenVMS VAX routines EXE$FINISHIO and EXE$FINISHIOC differ only in that the latter routine clears the second longword on I/O status, there is no OpenVMS AXP Step 2 equivalent of EXE$FINISHIOC. If the driver needs to clear the second I/O status longword, it simply does so before calling EXE_STD$FINISHIO.
- The address of the PCB, supplied as input to EXE$FINISHIO on OpenVMS VAX systems, is not provided as input to EXE_STD$FINISHIO.
- Unlike EXE$FINISHIO, EXE_STD$FINISHIO does not lower IPL to 0 before exiting. EXE_STD$FINISHIO returns to its caller at the caller's IPL.
- EXE$FINISHIO returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$FINISHIO returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE$ILLIOFUNC

Aborts I/O preprocessing for an I/O function not supported a driver.

Module

SYSQIOFDT

Format

status = EXE$ILLIOFUNC (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request

pcb
Process control block of the current process

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Context

FDT dispatching code in the $QIO system service calls EXE$ILLIOFUNC at IPL$_ASTDEL when processing an I/O function that is not supported by a driver. EXE$ILLIOFUNC returns to the system service dispatcher at IPL$_ASTDEL.

Description

Because any slot corresponding to an unsupported function in a driver's FDT action vector contains the procedure value of EXE$ILLIOFUNC, FDT dispatching code in the $QIO system service calls EXE$ILLIOFUNC to process any I/O request specifying an unsupported I/O function code.

EXE$ILLIOFUNC calls EXE_STD$ABORTIO to terminate the processing of the I/O request.

EXE_STD$INSERT_IRP

Inserts an I/O request packet (IRP) into the specified queue of IRPs according to the base priority of the process that issued the I/O request.

Module

SYSQIOREQ

Format

status = EXE_STD$INSERT_IRP (irp_lh, irp)

Arguments

Argument	Type	Access	Mechanism	Status
queue	listhead	input	reference	required
irp_lh	address	input	reference	required

irp_lh
I/O queue listhead for the device.

irp
I/O request packet. EXE_STD$INSERT_IRP reads the base address of the process requesting the I/O from IRP$B_PRI.

Return Values

status	Low bit set if at least one IRP is already in the queue, low bit clear if the IRP is the only entry.

Context

EXE_STD$INSERT_IRP must be called at fork IPL or higher. In an OpenVMS multiprocessing environment, the caller must also hold the associated fork lock. EXE_STD$INSERT_IRP does not alter IPL or acquire any spin locks. It returns to its caller.

Description

EXE_STD$INSERT_IRP determines the position of the specified IRP in the pending-I/O queue according to two factors:

- Priority of the IRP, which is derived from the requesting process's base priority as stored in the IRP$B_PRI
- Time that the entry is queued; for each priority, the queue is ordered on a first-in/first-out basis

EXE_STD$INSERT_IRP inserts the IRP into the queue at that position, adjusts the queue links, and returns a value to indicate the status of the queue.

Macro

$INSERT_IRP

In a Step 2 OpenVMS AXP driver, the $INSERT_IRP macro simulates a JSB to EXE$INSERT_IRP in a Step 1 OpenVMS AXP driver. $INSERT_IRP calls EXE_STD$INSERT_IRP, using the current contents of R2 and R3 as the **irp_lh** and **irp** arguments, respectively. It returns status in R0.

Notes for Converting Step 1 Drivers

- EXE_STD$INSERT_IRP replaces EXE$INSERT_IRP (used by Step 1 OpenVMS AXP drivers) and EXE$INSERTIRP (used by OpenVMS VAX drivers).

EXE_STD$INSIOQ, EXE_STD$INSIOQC

Insert an IRP in a device's pending-I/O queue and call the driver's start-I/O routine if the device is not busy.

Module

SYSQIOREQ

Format

EXE_STD$INSIOQ (irp, ucb)

EXE_STD$INSIOQC (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet.

ucb
Unit control block.

EXE_STD$INSIOQ and EXE_STD$INSIOQC read the following UCB fields:

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_BSY set if device is busy, clear if device is idle
UCB$L_IOQFL	Address of pending-I/O queue listhead
UCB$L_QLEN	Length of pending-I/O queue

EXE_STD$INSIOQ and EXE_STD$INSIOQC write the following UCB fields:

Field	Contents
UCB$L_STS	UCB$V_BSY set
UCB$W_QLEN	Incremented

Context

EXE_STD$INSIOQ and EXE_STD$INSIOQC immediately raise to fork IPL and, in a multiprocessing environment, obtain the corresponding fork lock. As a result, their callers must not be executing at an IPL higher than fork IPL or hold a spin lock ranked higher than the fork lock.

EXE_STD$INSIOQ unconditionally releases ownership of the fork lock before returning control to the caller without possession of the fork lock. If a fork process must retain possession of the fork lock, it should call EXE_STD$INSIOQC instead.

Description

EXE_STD$INSIOQ and EXE_STD$INSIOQC insert an IRP in a device's pending-I/O queue and call the driver's start-I/O routine if the device is not busy.

EXE_STD$INSIOQ and EXE_STD$INSIOQC increment UCB$L_QLEN and proceed according to the status of the device (as indicated by UCB$V_BSY in UCB$L_STS) as follows:

- If the device is busy, call EXE_STD$INSERT_IRP to place the IRP on the device's pending-I/O queue.
- If the device is idle, call IOC_STD$INITIATE to begin device processing of the I/O request immediately. IOC_STD$INITIATE transfers control to the driver's start-I/O routine.

Macro

$INSIOQ
$INSIOQC

In a Step 2 OpenVMS AXP driver, the $INSIOQ and $INSIOQC macros simulate a JSB to EXE$INSIOQ and EXE$INSIOQ, respectively, in a Step 1 driver. $INSIOQ calls EXE_STD$INSIOQ, and $INSIOQC calls EXE_STD$INSIOQC, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

None.

EXE_STD$IORSNWAIT

Places a process in a resource wait state if it has enabled resource waits.

Module

SYSQIOFDT

Format

status = EXE_STD$IORSNWAIT (irp, pcb, ucb, ccb, qio_sts, rsn)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
qio_sts	input	integer	value	required
rsn	input	integer	value	required

irp
I/O request packet.

pcb
Process control block.

ucb
Unit control block.

ccb
Channel control block.

qio_sts
Final status to be returned by the $QIO system service to its caller if the caller has not enabled resource wait mode. EXE_STD$IORSNWAIT calls EXE_STD$ABORTIO to place this status in FDT_CONTEXT$L_QIO_STATUS. If you intend to access the FDT context structure after EXE_STD$IORSNWAIT returns, you must obtain its address from IRP$PS_FDT_CONTEXT and store it before making the call.

rsn
Number of the resource for which the request is waiting.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

Contents of **qio_sts** argument	Process has not enabled resource waits.
SS$_WAIT_CALLERS_MODE	Process has been placed in a resource wait state.

Context

EXE_STD$IORSNWAIT is called by, and returns to, a driver's FDT routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

```
$IORSNWAIT [do_ret=YES]
```

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In a Step 2 OpenVMS AXP driver, the $IORSNWAIT macro simulates the JMP to EXE$IORSNWAIT in the FDT routine of a Step 1 driver. It calls EXE_STD$IORSNWAIT using the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **qio_sts**, and **rsn** arguments, respectively. When EXE_STD$IORSNWAIT returns control to the code generated by a default invocation of $IORSNWAIT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- EXE_STD$IORSNWAIT replaces EXE$IORSNWAIT (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). The order in which formal parameters are passed to EXE_STD$IORSNWAIT differs from the order in which they are provided in registers to the Step 1 OpenVMS routine EXE$IORSNWAIT.
- EXE$IORSNWAIT returns control to the system service dispatcher, passing it the final $QIO system service status in R0. EXE_STD$IORSNWAIT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$LCLDSKVALID

Processes I/O functions that affect the online count and local valid status of a disk.

Module

SYSQIOFDT

Format

status = EXE_STD$LCLDSKVALID (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp

I/O request packet for the current I/O request. The I/O function for the current request is available in IRP$L_FUNC.

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

EXE_STD$LCLDSKVALID reads the following UCB fields.

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_LCL_VALID set if the volume is valid; clear if the drive is unloaded or available
UCB$B_ONLCNT	Number of hosts that have set this disk on line

EXE_STD$LCLDSKVALID writes the following UCB fields:

Field	Contents
UCB$L_STS	UCB$V_LCL_VALID set if the requested function is IO$_PACKACK; cleared if the requested function is IO$_UNLOAD or IO$_AVAILABLE

Field	Contents
UCB$B_ONLCNT	Incremented if UCB$V_LCL_VALID is not set and the requested function is IO$_PACKACK; decremented if UCB$V_LCL_VALID is set and the requested function is IO$_UNLOAD or IO$_AVAILABLE

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code calls EXE_STD$LCLDSKVALID at IPL$_ASTDEL. EXE_STD$LCLDSKVALID immediately raises IPL to fork IPL, requesting the associated fork lock in a multiprocessing environment. When it regains control from EXE_STD$QIODRVPKT or EXE_STD$FINISHIO, EXE_STD$LCLDSKVALID lowers IPL to IPL$_ASTDEL and relinquishes the fork lock before returning to the system service dispatcher.

Description

A disk driver specifies the system-supplied upper-level FDT action routine EXE_STD$LCLDSKVALID in an FDT_ACT macro invocation to service a request for an IO$_PACKACK, IO$_AVAILABLE, or IO$_UNLOAD function for a local disk. The actions of EXE_STD$LCLDSKVALID depend on the I/O function indicated by R7 and the value of UCB$V_LCL_VALID in UCB$L_STS.

For an IO$_PACKACK function, EXE_STD$LCLDSKVALID proceeds as follows:

- If UCB$V_LCL_VALID is clear –
  - Sets UCB$V_LCL_VALID.
  - Increments UCB$B_ONLCNT.
  - If this is the first cluster pack acknowledgement on the disk (that is, if UCB$B_ONLCNT equals 1), invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$LCLDSKVALID regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.
- If UCB$V_LCL_VALID is set, EXE_STD$LCLDSKVALID requests that the FDT completion routine EXE_STD$FINISHIO complete the I/O request. EXE_STD$FINISHIO returns to EXE_STD$LCLDSKVALID with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. and return SS$_NORMAL status as the final $QIO system service status.

For an IO$_UNLOAD or IO$_AVAILABLE function, EXE_STD$LCLDSKVALID proceeds as follows:

- If UCB$V_LCL_VALID is set –
  - Clears UCB$V_LCL_VALID
  - Decrements UCB$B_ONLCNT
  - If this is the last cluster unload or available request, invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$LCLDSKVALID regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.
- If UCB$V_LCL_VALID is clear, EXE_STD$LCLDSKVALID requests that the FDT completion routine EXE_STD$FINISHIO complete the I/O request. EXE_STD$FINISHIO returns to EXE_STD$LCLDSKVALID with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

Note that a driver must define the local disk UCB extension, to use this routine.

Macro

None.

Notes for Converting Step 1 Drivers

- The upper-level FDT routine EXE$LCLDSKVALID (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$LCLDSKVALID.

EXE_STD$MNTVERSIO

Initiates a mount verification I/O request to a device.

Module

MOUNTVER

Format

EXE_STD$MNTVERSIO (rout, irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
rout	procedure value	input	value	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

rout
Procedure value of action routine to postprocess the mount verification I/O request.

irp
I/O request packet.

ucb
Unit control block.

Context

EXE_STD$MNTVERSIO raises IPL to fork IPL, obtaining the corresponding fork lock in an OpenVMS multiprocessing system. It releases the fork lock and returns control to its caller at its caller's IPL.

Description

For Digital internal use only.

Macro

$MNTVERSIO

In a Step 2 OpenVMS AXP driver, the $MNTVERSIO macro simulates a JSB to EXE$MNTVERSIO in a Step 1 OpenVMS AXP driver. $MNTVERSIO calls EXE_STD$MNTVERSIO, using the current contents of R0, R3, and R5 as the **rout**, **irp**, and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- EXE_STD$MNTVERSIO replaces EXE$MNTVERSIO (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers).

EXE_STD$MODIFY

Translates a logical read/write function into a physical read/write function, transfers $QIO system service parameters to the IRP, validates and prepares a user buffer, and aborts the request or proceeds with a direct-I/O, DMA read/write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$MODIFY (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp

I/O request packet for the current I/O request.

EXE_STD$MODIFY reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the buffer's virtual address.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the number of bytes in transfer. The maximum number of bytes that EXE_STD$MODIFY can transfer is 65,535 (128 pages minus one byte).
IRP$L_QIO_P4	$QIO system service **p4** argument, containing the carriage control byte.
IRP$L_FUNC	I/O function code.
IRP$B_RMOD	Access mode of the caller of the $QIO system service.

EXE_STD$MODIFY writes the following IRP fields:

Field	Contents
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical read/write function code converted to physical

Field	Contents
IRP$L_STS	IRP$V_FUNC set to indicate read function
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buffer** parameter does not allow read access.
SS$_BADPARAM	**size** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	The I/O request has been successfully queued.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$MODIFY as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$MODIFY to prepare a direct-I/O read/write request. A driver cannot specify EXE_STD$MODIFY for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own upper-level FDT action routines to handle them.

EXE_STD$MODIFY performs the following functions:

- Sets IRP$V_FUNC in IRP$L_STS to indicate a read function
- Copies the **p4** argument of the $QIO request from IRP$L_QIO_P4 to IRP$B_CARCON

- Translates a logical read/write function to a physical read/write function and stores the new function code in IRP$L_FUNC.
- Examines the size of the transfer, as specified in the **p2** argument of the $QIO request (IRP$L_QIO_P2), and takes one of the following actions:
  - If the transfer byte count is zero, EXE_STD$MODIFY invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$MODIFY regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

    The driver start-I/O routine should check for zero-length buffers to avoid mapping to adapter node space. An attempted mapping can cause a system failure.
  - If the byte count is not zero, EXE_STD$MODIFY calls EXE_STD$MODIFYLOCK, passing 0 as the value of the **err_rout** argument.

EXE_STD$MODIFYLOCK disables an optimization in MMG_STD$IOLOCK and joins the code for EXE_STD$READLOCK. EXE_STD$MODIFYLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**size** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$READCHK returns to EXE_STD$MODIFYLOCK with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFYLOCK immediately returns to EXE_STD$MODIFY, passing these status values. EXE_STD$MODIFY, in turn, returns to the $QIO system service.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access returns SS$_NORMAL in R0 to EXE_STD$MODIFYLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$READCHK returns to EXE_STD$MODIFYLOCK with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFYLOCK immediately returns to EXE_STD$MODIFY, passing these status values. EXE_STD$MODIFY returns to the $QIO system service.

If EXE_STD$READCHK succeeds, EXE_STD$MODIFYLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$MODIFYLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$MODIFYLOCK. EXE_STD$MODIFYLOCK returns immediately to EXE_STD$MODIFY, passing to it this status value.

EXE_STD$MODIFY invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$MODIFY regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$MODIFYLOCK.

  For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$MODIFYLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$MODIFYLOCK returns EXE_STD$MODIFY the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFY returns to the $QIO system service.

  For page fault status, EXE_STD$MODIFYLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine EXE$MODIFY (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$MODIFY.

- EXE$MODIFY returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_BADPARAM, or SS$_INSFWSL) in R0. EXE_STD$MODIFY returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$MODIFYLOCK

Validates and prepares a user buffer for a direct-I/O, DMA read/write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$MODIFYLOCK (irp, pcb, ucb, ccb, buf, bufsiz, err_rout)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required
err_rout	procedure value	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$MODIFYLOCK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$MODIFYLOCK writes the following IRP fields:

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

err_rout
Procedure value of error-handling callback routine, or 0 if the driver does not process errors.

A driver typically specifies an error-handling callback routine when the driver must lock multiple areas into memory for a single I/O request and regain control to unlock these areas, if the request is to be aborted. The routine performs those tasks required before the request is backed out of or aborted. Such operations could include calling MMG_STD$UNLOCK to release previous buffers participating in the I/O operation. The error-handling routine must preserve R0 and R1 and return back to EXE_STD$MODIFYLOCK.

Chapter 3 describes the error-handling callback routine interface.

Return Values

SS$_NORMAL	The buffer is read-accessible and has been locked in memory.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.
SS$_INSFWSL	Insufficient working set limit.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

The system-supplied upper-level FDT action routine EXE_STD$MODIFY, or a driver-specific upper-level FDT action routine, calls EXE_STD$MODIFYLOCK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$MODIFYLOCK to check the read accessibility of an I/O buffer supplied in a $QIO request for a read/write function, and lock the buffer in memory in preparation for a DMA read/write operation.

A driver cannot specify EXE_STD$MODIFY for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their FDT routines to handle them.

EXE_STD$MODIFYLOCK disables an optimization in MMG_STD$IOLOCK and joins the code for EXE_STD$READLOCK. EXE_STD$MODIFYLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, EXE_STD$READCHK returns SS$_BADPARAM status to EXE_STD$MODIFYLOCK.

- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to EXE_STD$MODIFYLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK returns SS$_ACCVIO status to EXE$_STD$MODIFYLOCK.

If error status (SS$_BADPARAM or SS$_ACCVIO) is returned, EXE_STD$MODIFYLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$MODIFYLOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$MODIFYLOCK calls EXE_STD$ABORTIO, passing it the error status as **qio_sts**. EXE_STD$ABORTIO returns to EXE_STD$MODIFYLOCK with the error status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFYLOCK immediately returns to its caller, passing these status values.

If SS$_NORMAL status is returned, EXE_STD$MODIFYLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$MODIFYLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$MODIFYLOCK. EXE_STD$MODIFYLOCK returns immediately to its caller, passing to it this status value.
- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$MODIFYLOCK. EXE_STD$MODIFYLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$MODIFYLOCK. When

the callback routine returns (or if no callback routine is specified), EXE_STD$MODIFYLOCK proceeds as follows:

- For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$MODIFYLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$MODIFYLOCK returns to its caller the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

  For page fault status, EXE_STD$MODIFYLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

The caller of EXE_STD$MODIFYLOCK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write accessible and has been successfully locked into memory and the starting virtual address of the page table entries that map the buffer is available in IRP$L_SVAPTE.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS. Ordinarily a driver specifies an error-handling callback routine to process such errors.

  Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$MODIFYLOCK.

Macro

```
$MODIFYLOCK
$MODIFYLOCK_ERR [interface_warning=YES]
```

where:

interface_warning=YES, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

In a Step 2 OpenVMS AXP driver, $MODIFYLOCK simulates a JSB to EXE$MODIFYLOCK and $MODIFYLOCK_ERR simulates a JSB to EXE$MODIFYLOCK_ERR. $MODIFYLOCK calls EXE_STD$MODIFYLOCK, specifying 0 as the **err_rout** argument; $MODIFYLOCK_ERR also calls EXE_STD$MODIFYLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$MODIFYLOCK or EXE_STD$MODIFYLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and writes a 5 into R2 to indicate a modify operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 5 to R2 to indicate a modify operation and and returns to FDT dispatching code in the $QIO system service.

Notes for Converting Step 1 Drivers

- EXE_STD$MODIFYLOCK replaces EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR (used by Step 1 OpenVMS AXP drivers). For compatibility with the Step 1 routines, use the $MODIFYLOCK and $MODIFYLOCK_ERR macros.
- EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$MODIFYLOCK.
- The order in which formal parameters are passed to EXE_STD$MODIFYLOCK differs from the order in which they are provided in registers to the Step 1 OpenVMS routines EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR.
- EXE$MODIFYLOCK_ERR provides a mechanism by which a driver callback routine obtains control upon an error condition prior to the abortion of an I/O request. EXE_STD$MODIFYLOCK accepts the address of an error-handling callback routine in the **err_rout** argument. The error-handling routine is called after an I/O request encounters a buffer access or memory allocation failure and before the request is aborted.
- The design of FDT processing for Step 2 OpenVMS AXP device drivers guarantees that the caller of EXE_STD$MODIFYLOCK regains control whether the modify lock operation is successful or not. When a driver regains control from a call to EXE_STD$MODIFYLOCK, return status in R0 indicates that the buffer has been successfully locked (SS$_NORMAL) or that the operation failed and the request has been aborted (SS$_FDT_COMPL). The driver must check the return status and take appropriate action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

  Normally, a driver services a modify lock failure by supplying the address of an error-handling callback routine to EXE_STD$MODIFYLOCK.
- Driver code that executes after receiving failure status (SS$_FDT_COMPL) from EXE_STD$MODIFYLOCK cannot access information in the IRP. If the driver anticipates accessing IRP fields when EXE_STD$MODIFYLOCK returns, it must store these fields elsewhere before calling EXE_STD$MODIFYLOCK.

- Upon successful completion, EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR provide as output the system virtual address of the first process PTE that maps the buffer in R1 and in IRP$L_SVAPTE. Because EXE_STD$MODIFYLOCK does not provide R1 as output, a driver must obtain this information from IRP$L_SVAPTE. Similarly, the Step 1 routines set R2 to 1 to indicate a a read function. EXE_STD$MODIFYLOCK does not provide R2 as output; a driver can determine whether a function is write or read by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$MOUNT_VER

During I/O postprocessing, determines whether mount verification should be initiated on a given disk or tape device on behalf of the I/O request being completed.

Module

MOUNTVER

Format

status = EXE_STD$MOUNT_VER (iost1, iost2, irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
iost1	integer	input	value	required
iost2	integer	input	value	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

iost1
First longword of I/O status.

iost2
Second longword of I/O status.

irp
I/O request packet. This argument is 0 if there is no IRP to clean up.

ucb
Unit control block.

Return Values

status	Low bit set indicates that mount verification has not been initiated and that the caller should continue; low bit clear indicates that mount verification has been initiated and that the caller should return.

Context

EXE_STD$MOUNT_VER is typically called at or above IPL$_IOPOST.

Description

For Digital internal use only.

Macro

$MOUNT_VER [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to EXE_STD$MOUNT_VER. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, the $MOUNT_VER macro simulates a JSB to EXE$MOUNT_VER in a Step 1 OpenVMS driver. $MOUNT_VER calls EXE_STD$MOUNT_VER, using the current contents of R0, R1, R3, and R5 as the **iost1**, **iost2**, **irp**, and **ucb** arguments, respectively. When EXE_STD$MOUNT_VER returns, code generated by this macro copies return status from R0 to R2. Unless you specify **save_r0r1=NO**, the macro preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- EXE_STD$MOUNT_VER replaces EXE$MOUNT_VER (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike EXE$MOUNT_VER, EXE_STD$MOUNT_VER does not preserve R0 and R1 across the call, or provide its return status in R2.

EXE_STD$ONEPARM

Copies a single $QIO parameter from IRP$L_QIO_P1 to IRP$L_MEDIA and delivers the IRP to a driver's start-I/O routine.

Module

SYSQIOFDT

Format

status = EXE_STD$ONEPARM (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request. EXE_STD$ONEPARM copies the first $QIO function-specific parameter (**p1**) from IRP$L_QIO_P1 to IRP$L_MEDIA.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$ONEPARM as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$ONEPARM to process an I/O function code that requires only one parameter. This parameter should need no checking: for instance, for read or write accessibility.

EXE_STD$ONEPARM copies the first $QIO function-dependent parameter (**p1**) from IRP$L_QIO_P1 to IRP$L_MEDIA and invokes the $QIODRVPKT macro to deliver the IRP to the driver. EXE_STD$ONEPARM regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine EXE$ONEPARM (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) obtains the value of the first function-dependent argument (**p1**) specified in the $QIO request from 00(AP). An OpenVMS AXP FDT routine cannot obtain the argument as an offset from the AP; rather, it accesses it from a new IRP field, IRP$L_QIO_P1.

  In order allow an OpenVMS VAX driver to be more easily converted to an OpenVMS AXP driver, the upper-level action routine EXE_STD$ONEPARM exists, simply to move the value of (**p1**) from IRP$L_QIO_P1 to IRP$L_MEDIA and invokes the $QIODRVPKT macro. If your driver does not access (**p1**) from IRP$L_MEDIA, but rather, uses the contents of IRP$L_QIO_P1, specifying EXE_STD$ONEPARM as an upper-level FDT action routine may defy all logic. (If your driver ignores the contents of IRP$L_MEDIA, it is immaterial whether you specify EXE_STD$ZEROPARM or EXE_STD$ONEPARM as the upper-level FDT action routine that delivers the IRP to the driver.) If you want to avoid the unnecessary copy, you can write an upper-level FDT action routine that simply invokes the $QIODRVPKT macro.

- EXE$ONEPARM expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$ONEPARM.

- EXE$ONEPARM returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$ONEPARM returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$PRIMITIVE_FORK

Creates a simple fork process on the local processor.

Module

FORKCNTRL

Format

EXE_STD$PRIMITIVE_FORK (fr3, fr4, fkb)

Arguments

Argument	Type	Access	Mechanism	Status
fr3	int64	input	value	required
fr4	int64	input	value	required
fkb	FKB	input	reference	required

fr3
Value to pass to the fork routine in FKB$Q_FR3.

fr4
Value to pass to the fork routine in FKB$Q_FR4.

fkb
Fork block. At input, FKB$B_FLCK must contain the fork lock index and FKB$L_FPC must contain the procedure value of the fork routine.

Context

EXE_STD$PRIMITIVE_FORK acquires no spin locks and leaves IPL unchanged. EXE_STD$PRIMITIVE_FORK, unlike the OpenVMS VAX system routine EXE$FORK, returns to its caller and not to its caller's caller. It assumes that, prior to the call, its caller has placed the procedure value of the fork routine into FKB$L_FPC.

EXE_STD$PRIMITIVE_FORK provides fork context to the fork routine in FKB$Q_FR3 (contents of **fr3**) and FKB$Q_FR4 (contents **fr4**). All other registers are destroyed. The fork routine executes at the IPL indicated by the fork lock index stored in FKB$B_FLCK.

Description

EXE_STD$PRIMITIVE_FORK moves the contents of the **fr3** and **fr4** arguments into FKB$Q_FR3 and FKB$Q_FR4, respectively. It determines the fork IPL by using the value of FKB$B_FLCK as an index into the spin lock IPL vector (SMP$AL_IPLVEC). EXE_STD$PRIMITIVE_FORK inserts the fork block into the fork queue on the local processor (headed by CPU$Q_SWIQFL) corresponding to this IPL. If the queue is empty, EXE_STD$PRIMITIVE_FORK issues a SOFTINT macro, requesting a software interrupt from the local processor at that fork IPL.

A driver that calls EXE_STD$PRIMITIVE_FORK explicitly (that is, instead of invoking the IOFORK macro) must ensure that UCB$V_TIM in the UCB$L_STS field is clear before making the call.

Macro

FORK

IOFORK

Notes for Converting Step 1 Drivers

- EXE_STD$PRIMITIVE_FORK is a call-based routine that performs the same operation as the JSB-based routine EXE$PRIMITIVE_FORK on OpenVMS AXP systems. The OpenVMS VAX routines EXE$FORK and EXE$IOFORK are not provided on OpenVMS AXP systems.

EXE_STD$PRIMITIVE_FORK_WAIT

Inserts a fork block on the fork-and-wait queue.

Module

FORKCNTRL

Format

EXE_STD$PRIMITIVE_FORK_WAIT (fr3, fr4, fkb)

Arguments

Argument	Type	Access	Mechanism	Status
fr3	int64	input	value	required
fr4	int64	input	value	required
fkb	FKB	input	reference	required

fr3
Value to pass to the fork routine in FKB$Q_FR3.

fr4
Value to pass to the fork routine in FKB$Q_FR4.

fkb
Fork block. At input, FKB$B_FLCK must contain the fork lock index and FKB$L_FPC must contain the procedure value of the fork routine.

Context

The caller of EXE_STD$PRIMITIVE_FORK_WAIT must be executing at or above IPL$_SYNCH. EXE_STD$PRIMITIVE_FORK_WAIT acquires the MEGA (SPL$C_MEGA) spin lock, raising IPL to IPL$_MEGA in the process, to access the fork-and-wait queue (EXE$AR_FORK_WAIT_QUEUE). It releases the spin lock, restoring the previous IPL, prior to returning to its caller.

EXE_STD$PRIMITIVE_FORK_WAIT, unlike the OpenVMS VAX system routine EXE$FORK_WAIT, returns to its caller and not to its caller's caller. It assumes that, prior to the call, its caller has placed the procedure value of the fork routine into FKB$L_FPC.

EXE_STD$PRIMITIVE_FORK_WAIT provides fork context to the fork routine in FKB$Q_FR3 (contents of **fr3**) and FKB$Q_FR4 (contents of **fr4**). All other registers are destroyed. The fork routine executes at the IPL indicated by the fork lock index stored in FKB$B_FLCK.

Description

EXE_STD$PRIMITIVE_FORK_WAIT moves the contents of **fr3** and **fr4** into FKB$Q_FR3 and FKB$Q_FR4 respectively. Having obtained the MEGA spin lock, it inserts the fork block indicated by **fkb** at end of the fork-and-wait queue (EXE$GL_FKWAITBL) and releases the spin lock.

Up to one second later, the software timer interrupt service routine will remove this and all other entries from the fork-and-wait queue and resume their respective fork routines.

Macro

FORK_WAIT

Notes for Converting Step 1 Drivers

- EXE_STD$PRIMITIVE_FORK_WAIT is a call-based routine that performs the same operation as the JSB-based routine EXE$PRIMITIVE_FORK_WAIT on OpenVMS AXP systems. The OpenVMS VAX routines EXE$FORK and EXE$IOFORK are not provided on OpenVMS AXP systems.

EXE_STD$QIOACPPKT

Delivers an IRP to the appropriate ACP or XQP.

Module

SYSQIOREQ

Format

status = EXE_STD$QIOACPPKT (irp, pcb, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet.

pcb
Process control block.

ucb
Unit control block.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$QIOACPPKT is called by, and returns to, a driver's FDT routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$QIOACPPKT [do_ret=YES]

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In a Step 2 OpenVMS AXP driver, the $QIOACPPKT macro simulates the JMP to EXE$QIOACPPKT in the FDT routine of a Step 1 driver. It calls EXE_STD$QIOACPPKT using the current contents of R3, R4, and R5 as the **irp**, **pcb**, and **ucb** arguments, respectively. When EXE_STD$QIOACPPKT returns control to the code generated by a default invocation of $QIOACPPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- EXE$QIOACPPKT returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$QIOACPPKT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$QIODRVPKT

Delivers an IRP to a driver's start-I/O routine or pending-I/O queue.

Module

SYSQIOREQ

Format

status = EXE_STD$QIODRVPKT (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet. EXE_STD$QIODRVPKT clears IRP$PS_FDT_CONTEXT. The caller of EXE_STD$QIODRVPKT should not access the IRP after the routine returns SS$_FDT_COMPL status.

ucb
Unit control block.

EXE_STD$QIODRVPKT (by means of the call to EXE_STD$INSIOQ) reads the following UCB fields:

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_BSY set if device is busy, clear if device is idle
UCB$L_IOQFL	Address of pending-I/O queue listhead
UCB$L_QLEN	Length of pending-I/O queue

EXE_STD$QIODRVPKT (by means of the call to EXE_STD$INSIOQ) writes the following UCB fields:

Field	Contents
UCB$L_STS	UCB$V_BSY set
UCB$W_QLEN	Incremented

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$QIODRVPKT is called by—and returns to—a driver's FDT routine at IPL$_ASTDEL.

Description

The FDT completion routine EXE_STD$QIODRVPKT delivers an IRP to the driver's start-I/O routine or pending-I/O queue.

EXE_STD$QIODRVPKT clears the pointer to the FDT context structure in IRP$PS_FDT_CONTEXT and calls. EXE_STD$INSIOQ checks the status of the device and calls either EXE_STD$INSERT_IRP or IOC_STD$INITIATE to place the IRP in the device's pending-I/O queue or deliver it to the driver's start-I/O routine, respectively.

When EXE_STD$INSIOQ returns, EXE_STD$QIODRVPKT returns SS$_FDT_COMPL status to its caller, passing SS$_NORMAL as the final $QIO completion status in the FDT context structure.

The image that requested the I/O operation receives SS$_NORMAL status, indicating that the I/O request has completed without device-independent error.

Macro

```
$QIODRVPKT [do_ret=YES]
```

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In a Step 2 OpenVMS AXP driver, the $QIODRVPKT macro simulates the JMP to EXE$QIODRVPKT in the FDT routine of a Step 1 driver. $QIODRVPKT clears IRP$PS_FDT_CONTEXT and calls EXE_STD$INSIOQ, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. When EXE_STD$INSIOQ returns control to the code generated by a default invocation of $QIODRVPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0.

Notes for Converting Step 1 Drivers

- The address of the PCB, supplied as input to EXE$QIODRVPKT on OpenVMS VAX systems, is not provided as input to EXE_STD$QIODRVPKT.
- Unlike EXE$QIODRVPKT, EXE_STD$QIODRVPKT does not lower IPL to 0 before exiting. EXE_STD$QIODRVPKT returns to its caller at the caller's IPL.
- EXE$QIODRVPKT returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$QIODRVPKT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$QUEUE_FORK

TBS

EXE_STD$QXQPPKT

Inserts an IRP on the tail of the XQP work queue and initiates its processing if it is the only request on the queue.

Module

SYSQIOREQ

Format

status = EXE_STD$QXQPPKT (pcb, acb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required
acb	ACB	input	reference	required

pcb
Process control block.

acb
AST control block within the IRP.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$QXQPPKT is called by, and returns to, a driver's FDT routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$QXQPPKT

In a Step 2 OpenVMS AXP driver, the $QXQPPKT macro simulates the JMP to EXE$QXQPPKT in the FDT routine of a Step 1 driver. It calls EXE_STD$QXQPPKT using the current contents of R4 and R5 as the **pcb** and **acb** arguments, respectively. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting Step 1 Drivers

- EXE$QXQPPKT returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$QXQPPKT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$READ

Translates a logical read function into a physical read function, transfers $QIO system service parameters to the IRP, validates and prepares a user buffer, and aborts the request or proceeds with a direct-I/O, DMA write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$READ (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp

I/O request packet for the current I/O request.

EXE_STD$READ reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the buffer's virtual address.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the number of bytes in transfer. The maximum number of bytes that EXE_STD$READ can transfer is 65,535 (128 pages minus one byte).
IRP$L_QIO_P4	$QIO system service **p4** argument, containing the carriage control byte.
IRP$L_FUNC	I/O function code.
IRP$B_RMOD	Access mode of the caller of the $QIO system service.

EXE_STD$READ writes the following IRP fields:

Field	Contents
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical read function code converted to physical
IRP$L_STS	IRP$V_FUNC set to indicate read function

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow write access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	The I/O request has been successfully queued.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$READ as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$READ to prepare a direct-I/O read request. A driver cannot specify EXE_STD$READ for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own upper-level FDT action routines to handle them.

EXE_STD$READ performs the following functions:

- Sets IRP$V_FUNC in IRP$L_STS to indicate a read function
- Copies the **p4** argument of the $QIO request from IRP$L_QIO_P4 to IRP$B_CARCON

- Translates a logical read function to a physical read function and stores the new function code in IRP$L_FUNC.
- Examines the size of the transfer, as specified in the **p2** argument of the $QIO request (IRP$L_QIO_P2), and takes one of the following actions:
  - If the transfer byte count is zero, EXE_STD$READ invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$READ regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

    The driver start-I/O routine should check for zero-length buffers to avoid mapping to adapter node space. An attempted mapping can cause a system failure.
  - If the byte count is not zero, EXE_STD$READ calls EXE_STD$READLOCK, specifying 0 as the **err_rout** argument.

EXE_STD$READLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$READCHK returns to EXE_STD$READLOCK with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READLOCK immediately returns to EXE_STD$READ, passing these status values. EXE_STD$READ, in turn, returns to the $QIO system service.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to EXE_STD$READLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$READCHK returns to EXE_STD$READLOCK with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READLOCK immediately returns to EXE_STD$READ, passing these status values. EXE_STD$READ returns to the $QIO system service.

If EXE_STD$READCHK succeeds, EXE_STD$READLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$READLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$READLOCK. EXE_STD$READLOCK returns immediately to EXE_STD$READ, passing to it this status value.

EXE_STD$READ invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$READ regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$READLOCK.

  For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$READLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$READLOCK returns EXE_STD$READ the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READ returns to the $QIO system service.

  For page fault status, EXE_STD$READLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine EXE$READ (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$READ.

- EXE$READ returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_BADPARAM, or SS$_INSFWSL) in R0. EXE_STD$READ returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$READCHK

Verifies that a process has write access to the pages in the buffer specified in a $QIO request.

Module

SYSQIOFDT

Format

status = EXE_STD$READCHK (irp, pcb, ucb, buf, bufsiz)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required

irp

I/O request packet for the current I/O request.

EXE_STD$READCHK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$READCHK writes the following IRP fields:

Field	Contents
IRP$L_STS	IRP$V_FUNC set, indicating a read function
IRP$L_BCNT	Size of transfer in bytes

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

buf

Virtual address of buffer.

bufsiz

Number of bytes in transfer.

Return Values

SS$_NORMAL	The buffer is write-accessible.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow write access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.

Context

The FDT support routine EXE_STD$READLOCK, or a driver-specific FDT routine, calls EXE_STD$READCHK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$READCHK to check the write accessibility of an I/O buffer supplied in a $QIO request for a read function.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$READCHK returns to its caller with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to its caller.
  - If the buffer does not allow write access, EXE_STD$READCHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$READCHK returns to its caller with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

The caller of EXE_STD$READCHK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write-accessible.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS.

Certain drivers must perform additional processing to back out an I/O request after it has aborted. For instance, if the driver has locked multiple buffers into memory for a single I/O request, it must unlock them once the request has been aborted. Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$READCHK.

Macro

$READCHK
$READCHKR

In a Step 2 OpenVMS AXP driver, $READCHK simulates a JSB to EXE$READCHK and $READCHKR simulates a JSB to EXE$READCHKR. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$READCHK returns, $READCHK and $READCHKR move 1 into R2 to indicate a read operation and examines the return status:

- If success status (SS$_NORMAL) is returned, $READCHK and $READCHKR copy the contents of IRP$L_BCNT into R1. $READCHK writes the starting address of the I/O buffer in R0; $READCHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, $READCHK returns to FDT dispatching code in the $QIO system service. $READCHKR does not return control to $QIO.

Notes for Converting Step 1 Drivers

- EXE_STD$READCHK replaces EXE$READCHK and EXE$READCHKR (used by Step 1 OpenVMS drivers). For compatibility with the Step 1 routines, use the $READCHK and $READCHKR macros.
- EXE$READCHK and EXE$READCHKR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$READCHK.
- The order in which formal parameters are passed to EXE_STD$READCHK differs from the order in which they are provided in registers to the Step 1 routines EXE$READCHK and EXE$READCHKR.
- EXE$READCHK and EXE$READCHKR provide a mechanism by which a driver callback routine or coroutine obtains control upon an error condition prior to the abortion of an I/O request. The design of FDT processing for Step 2 OpenVMS AXP device drivers guarantees that the caller of EXE_STD$READCHK regains control whether the read check operation is successful. The caller must examine the return status in R0 (SS$_NORMAL indicates the buffer is write accessible, SS$_FDT_COMPL indicates that the operation failed and the request has been aborted) and take appropriate

action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

- Driver code that services failure status (SS$_FDT_COMPL) from EXE_STD$READLOCK (for instance a callback routine formerly specified to EXE$READLOCK_ERR) cannot access information in the IRP. If the driver anticipates handling failure status by using the contents of IRP fields, it must store these fields elsewhere before calling EXE_STD$READLOCK.

  This is especially important for driver code that expects EXE_STD$READCHK to access the transfer size in R1 after the call. Unlike EXE$READCHK and EXE$READCHKR, EXE_STD$READCHK does not preserve the contents of R1 and R3 across the call. If you must repeat a $READCHK macro invocation, you must be sure to reload R0, R1, and R3 with the virtual address of the buffer, the transfer size, and the address of the IRP, respectively, before each subsequent invocation.

- Upon successful completion, EXE$READCHK and EXE$READCHKR set R2 to 1 for a read function. EXE_STD$READCHK does not provide R2 as output; a driver can determine whether a function is read or write by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$READLOCK

Validates and prepares a user buffer for a direct-I/O, DMA write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$READLOCK (irp, pcb, ucb, ccb, buf, bufsiz, err_rout)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required
err_rout	procedure value	input	value	required

irp

I/O request packet for the current I/O request.

EXE_STD$READLOCK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$READLOCK writes the following IRP fields:

Field	Contents
IRP$L_STS	IRP$V_FUNC set, indicating a read function
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer

err_rout
Procedure value of error-handling callback routine, or 0 if the driver does not process errors.

A driver typically specifies an error-handling callback routine when the driver must lock multiple areas into memory for a single I/O request and regain control to unlock these areas, if the request is to be aborted. The routine performs those tasks required before the request is backed out of or aborted. Such operations could include calling MMG_STD$UNLOCK to release previous buffers participating in the I/O operation. The error-handling routine must preserve R0 and R1 and return back to EXE_STD$READLOCK.

Chapter 3 describes the error-handling callback routine interface.

Return Values

SS$_NORMAL	The buffer is write-accessible and has been locked in memory.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow write access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_ CONTEXT structure.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

The system-supplied upper-level FDT action routine EXE_STD$READ, or a driver-specific upper-level FDT action routine, calls EXE_STD$READLOCK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$READLOCK to check the write accessibility of an I/O buffer supplied in a $QIO request for a read function, and lock the buffer in memory in preparation for a DMA write operation.

A driver cannot specify EXE_STD$READ for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own FDT routines to handle them.

EXE_STD$READLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, EXE_STD$READCHK returns SS$_BADPARAM status to EXE_STD$READLOCK.

- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:

  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to EXE_STD$READLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK returns SS$_ACCVIO status to EXE$_STD$READLOCK.

If error status (SS$_BADPARAM or SS$_ACCVIO) is returned, EXE_STD$READLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$READLOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$READLOCK calls EXE_STD$ABORTIO, passing it the error status as **qio_sts**. EXE_STD$ABORTIO returns to EXE_STD$READLOCK with the error status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READLOCK immediately returns to its caller, passing these status values.

If SS$_NORMAL status is returned, EXE_STD$READLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$READLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$READLOCK. EXE_STD$READLOCK returns immediately to its caller, passing to it this status value.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$READLOCK. EXE_STD$READLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$READLOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$READLOCK proceeds as follows:

- For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$READLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$READLOCK returns to its caller the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.
- For page fault status, EXE_STD$READLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

The caller of EXE_STD$READLOCK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write accessible and has been successfully locked into memory and the starting virtual address of the page table entries that map the buffer is available in IRP$L_SVAPTE.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS. Ordinarily a driver specifies an error-handling callback routine to process such errors.

  Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$READLOCK.

Macro

$READLOCK
$READLOCK_ERR [interface_warning=YES]

where:

interface_warning=YES, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

In a Step 2 OpenVMS AXP driver, the $READLOCK simulates a JSB to EXE$READLOCK and $READLOCK_ERR simulates a JSB to EXE$READLOCK_ERR. $READLOCK calls EXE_STD$READLOCK, specifying 0 as the **err_rout** argument; $READLOCK_ERR also calls EXE_STD$READLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$READLOCK or EXE_STD$READLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro copies the contents of IRP$L_SVAPTE into R1 and writes a 1 to R2 to indicate a read operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 1 to R2 to indicate a read operation and returns to FDT dispatching code in the $QIO system service.

Notes for Converting Step 1 Drivers

- EXE_STD$READLOCK replaces EXE$READLOCK and EXE$READLOCK_ERR (used by Step 1 OpenVMS AXP drivers). For compatibility with the Step 1 routines, use the $READLOCK and $READLOCK_ERR macros.
- EXE$READLOCK and EXE$READLOCK_ERR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$READLOCK.
- The order in which formal parameters are passed to EXE_STD$READLOCK differs from the order in which they are provided in registers to the Step 1 routines EXE$READLOCK and EXE$READLOCK_ERR.
- EXE$READLOCK_ERR provides a mechanism by which a driver callback routine obtains control upon an error condition prior to the abortion of an I/O request. EXE_STD$READLOCK accepts the address of an error-handling callback routine in the **err_rout** argument. The error-handling routine is called after an I/O request encounters a buffer access or memory allocation failure and before the request is aborted.
- The design of FDT processing for Step 2 OpenVMS AXP device drivers guarantees that the caller of EXE_STD$READLOCK regains control whether the read lock operation is successful. When a driver regains control from a call to EXE_STD$READLOCK, return status in R0 indicates that the buffer has been successfully locked (SS$_NORMAL) or that the operation failed and the request has been aborted (SS$_FDT_COMPL). The driver must check the return status and take appropriate action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

  Normally, a driver services a read lock failure by supplying the address of an error-handling callback routine to EXE_STD$READLOCK.
- Driver code that executes after receiving failure status (SS$_FDT_COMPL) from EXE_STD$READLOCK cannot access information in the IRP. If the driver anticipates accessing IRP fields when EXE_STD$READLOCK returns, it must store these fields elsewhere before calling EXE_STD$READLOCK.
- Upon successful completion, EXE$READLOCK and EXE$READLOCK_ERR provide as output the system virtual address of the first process PTE that maps the buffer in R1 and in IRP$L_SVAPTE. Because EXE_STD$READLOCK does not provide R1 as output, a driver must obtain this information from IRP$L_SVAPTE. Similarly, the Step 1 routines set R2 to 1

for a read function and clear it otherwise. EXE_STD$READLOCK does not provide R2 as output; a driver can determine whether a function is read or write by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$SENSEMODE

Copies device-dependent characteristics from the device's UCB into the second longword of the I/O status block (IOSB) specified in a $QIO system service call, and completes the I/O operation successfully.

Module

SYSQIOFDT

Format

status = EXE_STD$SENSEMODE (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request. EXE_STD$SENSEMODE reads the device-dependent status stored in UCB$L_DEVDEPEND.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$SENSEMODE as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$SENSEMODE to process the sense-device-mode (IO$_SENSEMODE) and sense-device-characteristics (IO$_SENSECHAR) I/O functions.

EXE_STD$SENSEMODE loads the contents of UCB$L_DEVDEPEND into the second longword of the I/O status block (IOSB) specified in the original $QIO system service call. It then places SS$_NORMAL status into the FDT_CONTEXT structure and transfers control to EXE_STD$FINISHIO to insert the IRP in the local processor's I/O postprocessing queue.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine EXE$SENSEMODE (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$SENSEMODE.
- EXE$SENSEMODE returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$SENSEMODE returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$SETCHAR, EXE_STD$SETMODE

Write device-specific status and control information into the device's UCB and complete the I/O request (EXE_STD$SETCHAR); or write the information into the IRP and deliver the IRP to the driver's start-I/O routine (EXE_STD$SETMODE).

Module

SYSQIOFDT

Format

status = EXE_STD$SETCHAR (irp, pcb, ucb, ccb)

status = EXE_STD$SETMODE (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp

I/O request packet for the current I/O request.

EXE_STD$SETCHAR and EXE_STD$SETMODE read the following IRP fields:

Field	Contents
IRP$L_FUNC	I/O function code supplied in the $QIO request
IRP$B_RMOD	Mode of the $QIO caller
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the device characteristics quadword.

EXE_STD$SETMODE writes the following IRP fields:

Field	Contents
IRP$L_MEDIA	First longword of device characteristics
IRP$L_MEDIA+4	Second longword of device characteristics

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

EXE_STD$SETCHAR writes the following UCB fields:

Field	Contents
UCB$B_DEVCLASS	Byte 0 of device characteristics quadword
UCB$B_DEVTYPE	Byte 1 of device characteristics quadword
UCB$W_DEVBUFSIZ	Bytes 2 and 3 of device characteristics quadword
UCB$L_DEVDEPEND	Bytes 4 through 7 of device characteristics quadword

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.
SS$_ACCVIO	Process calling the $QIO system service with the IO$_SETMODE or IO$_SETCHAR function does not have read access to the quadword containing the new device characteristics.
SS$_ILLIOFUNC	IO$_SETMODE and IO$_SETCHAR functions are not legal for disk devices.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$SETCHAR and EXE_STD$SETMODE as upper-level FDT action routines at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$SETCHAR or EXE_STD$SETMODE to process the set-device-mode (IO$_SETMODE) and set-device-characteristics (IO$_SETCHAR) functions, respectively. If setting device characteristics requires device activity or synchronization with fork processing, the driver's FDT_ACT macro invocation *must* specifies EXE_STD$SETMODE. Otherwise, it can specify EXE_STD$SETCHAR.

EXE_STD$SETCHAR and EXE_STD$SETMODE examine the current value of UCB$B_DEVCLASS to determine whether the device permits the specified function. If the device class is disk (DC$_DISK), the routines place SS$_ILLIOFUNC status in the FDT_CONTEXT structure and transfer control to EXE_STD$ABORTIO to terminate the request.

EXE_STD$SETCHAR and EXE_STD$SETMODE then ensure that the process has read access to the quadword containing the new device characteristics. If it does not, the routines place SS$_ACCVIO status in the FDT_CONTEXT structure and transfer control to EXE_STD$ABORTIO to terminate the request.

If the request passes these checks, EXE_STD$SETCHAR and EXE_STD$SETMODE proceed as follows:

- EXE_STD$SETCHAR stores the specified characteristics in the UCB. For an IO$_SETCHAR function, the device type and class fields (UCB$B_DEVCLASS and UCB$B_DEVTYPE, respectively) receive the first word of data. For both IO$_SETCHAR and IO$_SETMODE functions, EXE_STD$SETCHAR writes the second word into the default-buffer-size field (UCB$W_DEVBUFSIZ) and the third and fourth words into the device-dependent-characteristics field (UCB$Q_DEVDEPEND).

  Finally, EXE_STD$SETCHAR stores normal completion status (SS$_NORMAL) in the FDT_CONTEXT structure and transfers control to the FDT completion routine EXE_STD$FINISHIO to insert the IRP in the local processor's I/O postprocessing queue. EXE_STD$FINISHIO returns to EXE_STD$SETCHAR with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

- EXE_STD$SETMODE stores the specified quadword of characteristics in IRP$L_MEDIA, places normal completion status (SS$_NORMAL) in the FDT_CONTEXT structure, and transfers control to FDT completion routine EXE_STD$QIODRVPKT to deliver the IRP to the driver's start-I/O routine. EXE_STD$QIODRVPKT returns to EXE_STD$SETMODE with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

The driver's start-I/O routine copies data from IRP$L_MEDIA and the following longword into UCB$W_DEVBUFSIZ, UCB$L_DEVDEPEND, and, if the I/O function is IO$_SETCHAR, UCB$B_DEVCLASS and UCB$B_DEVTYPE as well.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routines EXE$SETCHAR and EXE$SETMODE (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$SETCHAR and EXE_STD$SETMODE.

- EXE$SETCHAR and EXE$SETMODE return control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_ILLIOFUNC) in R0. EXE_STD$SETCHAR or EXE_STD$SETMODE returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$SNDEVMSG

Builds and sends a device-specific message to the mailbox of a system process, such as the job controller or OPCOM.

Module

MBDRIVER

Format

status = EXE_STD$SNDEVMSG (mb_ucb, msgtyp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
mb_ucb	MB_UCB	input	reference	required
msgtyp	integer	input	value	required
ucb	UCB	input	reference	required

mb_ucb

Mailbox UCB. (SYS$AR_JOBCTLMB contains the address of the job controller's mailbox; SYS$AR_OPRMBX contains the address of OPCOM's mailbox.)

msgtyp

Message type. OPCOM message types have the prefix OPC$_ and are defined by the $OPCMSG macro in SYS$LIBRARY:STARLET.MLB.

ucb

Device UCB. EXE_STD$SNDEVMSG reads the following UCB fields:

UCB$W_UNIT	Device unit number.
UCB$L_DDB	Address of device DDB. EXE_STD$SNDEVMSG constructs the device controller name from DDB$T_NAME and mailbox UCB fields.

Return Values

SS$_DEVNOTMBX	**mb_ucb** does not specify a mailbox UCB.
SS$_INSFMEM	The system is unable to allocate memory for the message.
SS$_MBFULL	The message mailbox is full of messages.
SS$_MBTOOSML	The message is too large for the mailbox.
SS$_NOPRIV	The caller lacks privilege to write to the mailbox.
SS$_NORMAL	Normal, successful completion.

Context

Because EXE_STD$SNDEVMSG raises IPL to IPL$_MAILBOX and obtains the MAILBOX spin lock in a multiprocessing environment, its caller cannot be executing above IPL$_MAILBOX. EXE_STD$SNDEVMSG returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

EXE_STD$SNDEVMSG builds a 32-byte message on the stack that includes the following information:

Bytes	Contents
0 and 1	Low word of **msgtyp** parameter
2 and 3	Device unit number (UCB$W_UNIT)
4 through 31	Counted string of device controller name, formatted as *node$controller* for clusterwide devices

EXE_STD$SNDEVMSG then calls EXE_STD$WRTMAILBOX to send the message to a mailbox.

Macro

$SNDEVMSG [save_r1]

where:

save_r1 indicates that the macro should preserve register R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, the $SNDEVMSG macro simulates a JSB to EXE$SNDEVMSG in a Step 1 OpenVMS driver. $SNDEVMSG calls EXE_STD$SNDEVMSG, using the current contents of R3, R4, and R5 as the **mb_ucb**, **msgtyp**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1=NO**, the macro preserves R1 across the call.

Notes for Converting Step 1 Drivers

- EXE_STD$SNDEVMSG replaces EXE$SNDEVMSG (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike EXE$SNDEVMSG, EXE_STD$SNDEVMSG does not preserve R1 across the call.

EXE_STD$WRITE

Translates a logical write function into a physical write function, transfers $QIO system service parameters to the IRP, validates and prepares a user buffer, and aborts the request or proceeds with a direct-I/O, DMA read operation.

Module

SYSQIOFDT

Format

status = EXE_STD$WRITE (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

EXE_STD$WRITE reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the buffer's virtual address.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the number of bytes in transfer. The maximum number of bytes that EXE_STD$WRITE can transfer is 65,535 (128 pages minus one byte).
IRP$L_QIO_P4	$QIO system service **p4** argument, containing the carriage control byte.
IRP$L_FUNC	I/O function code.
IRP$B_RMOD	Access mode of the caller of the $QIO system service.

EXE_STD$WRITE writes the following IRP fields:

Field	Contents
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical write function code converted to physical

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	The I/O request has been successfully queued.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$WRITE as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$WRITE to prepare a direct-I/O write request. A driver cannot specify EXE_STD$WRITE for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own upper-level FDT action routines to handle them.

EXE_STD$WRITE performs the following functions:

- Copies the **p4** argument of the $QIO request from IRP$L_QIO_P4 to IRP$B_CARCON
- Translates a logical write function to a physical write function and stores the new function code in IRP$L_FUNC.

- Examines the size of the transfer, as specified in the **p2** argument of the $QIO request (IRP$L_QIO_P2), and takes one of the following actions:
  - If the transfer byte count is zero, EXE_STD$WRITE invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$WRITE regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

    The driver start-I/O routine should check for zero-length buffers to avoid mapping to adapter node space. An attempted mapping can cause a system failure.
  - If the byte count is not zero, EXE_STD$WRITE calls EXE_STD$WRITELOCK, passing 0 as the value of the **err_rout** argument.

EXE_STD$WRITELOCK invokes the $WRITECHK macro, which calls EXE_STD$WRITECHK.

EXE_STD$WRITECHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$WRITECHK returns to EXE_STD$WRITELOCK with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITELOCK immediately returns to EXE_STD$WRITE, passing these status values. EXE_STD$WRITE, in turn, returns to the $QIO system service.
- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:
  - If the buffer allows read access returns SS$_NORMAL in R0 to EXE_STD$WRITELOCK.
  - If the buffer does not allow read access, EXE_STD$WRITECHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$WRITECHK returns to EXE_STD$WRITELOCK with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITELOCK immediately returns to EXE_STD$WRITE, passing these status values. EXE_STD$WRITE returns to the $QIO system service.

If EXE_STD$WRITECHK succeeds, EXE_STD$WRITELOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$WRITELOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$WRITELOCK. EXE_STD$WRITELOCK returns immediately to EXE_STD$WRITE, passing to it this status value.

EXE_STD$WRITE invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$WRITE regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$WRITELOCK.

  For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$WRITELOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$WRITELOCK returns EXE_STD$WRITE the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITE returns to the $QIO system service.

  For page fault status, EXE_STD$WRITELOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine EXE$WRITE (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$WRITE.

- EXE$WRITE returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_BADPARAM, or SS$_INSFWSL) in R0. EXE_STD$WRITE returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$WRITECHK

Verifies that a process has read access to the pages in the buffer specified in a $QIO request.

Module

SYSQIOFDT

Format

status = EXE_STD$WRITECHK (irp, pcb, ucb, buf, bufsiz)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$WRITECHK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$WRITECHK writes the size of the transfer in bytes to IRP$L_BCNT.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

Return Values

SS$_NORMAL	The buffer is read-accessible.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.

Context

The FDT support routine EXE_STD$WRITELOCK, or a driver-specific FDT routine, calls EXE_STD$WRITECHK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$WRITECHK to check the read accessibility of an I/O buffer supplied in a $QIO request for a write function.

EXE_STD$WRITECHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$WRITECHK returns to its caller with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:

  - If the buffer allows read access, EXE_STD$WRITECHK returns SS$_NORMAL in R0 to its caller.
  - If the buffer does not allow read access, EXE_STD$WRITECHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$WRITECHK returns to its caller with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

The caller of EXE_STD$WRITECHK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is read-accessible.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS.

Certain drivers must perform additional processing to back out an I/O request after it has aborted. For instance, if the driver has locked multiple buffers into memory for a single I/O request, it must unlock them once the request has been aborted. Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$WRITELOCK.

Macro

$WRITECHK
$WRITECHKR

In a Step 2 OpenVMS AXP driver, $WRITECHK simulates a JSB to EXE$WRITECHK and $READCHKR simulates a JSB to EXE$READCHKR. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$WRITECHK returns, $WRITECHK and $WRITECHKR clear R2 to indicate a write operation and examines the return status:

- If success status (SS$_NORMAL) is returned, $WRITECHK and $WRITECHKR copy the contents of IRP$L_BCNT into R1. $WRITECHK writes the starting address of the I/O buffer in R0; $WRITECHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, $WRITECHK returns to FDT dispatching code in the $QIO system service. $WRITECHKR does not return control to $QIO.

Notes for Converting Step 1 Drivers

- EXE_STD$WRITECHK replaces EXE$WRITECHK and EXE$WRITECHKR (used by Step 1 OpenVMS drivers). For compatibility with the Step 1 routines, use the $WRITECHK and $WRITECHKR macros.
- EXE$WRITECHK and EXE$WRITECHKR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$WRITECHK.
- The order in which formal parameters are passed to EXE_STD$WRITECHK differs from the order in which they are provided in registers to the Step 1 OpenVMS routines EXE$WRITECHK and EXE$WRITECHKR.
- EXE$WRITECHK and EXE$WRITECHKR provide a mechanism by which a driver callback routine or coroutine obtains control upon an error condition prior to the abortion of an I/O request. The design of FDT processing for Step 2 OpenVMS AXP device drivers guarantees that the caller of EXE_STD$WRITECHK regains control whether the write check operation is successful. The caller must examine the return status in R0 (SS$_NORMAL indicates the buffer is read accessible, SS$_FDT_COMPL indicates that the operation failed and the request has been aborted) and take appropriate

action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

- Driver code that services failure status (SS$_FDT_COMPL) from EXE_STD$WRITELOCK (for instance, a callback routine formerly specified to EXE$WRITELOCK_ERR) cannot access information in the IRP. If the driver anticipates handling failure status by using the contents of IRP fields, it must store these fields elsewhere before calling EXE_STD$WRITELOCK.

  This is especially important for driver code that expects EXE_STD$WRITECHK to access the transfer size in R1 after the call. Unlike EXE$WRITECHK and EXE$WRITECHKR, EXE_STD$WRITECHK does not preserve the contents of R1 and R3 across the call. If you must repeat a $WRITECHK macro invocation, be sure to reload R0, R1, and R3 with the virtual address of the buffer, the transfer size, and the address of the IRP, respectively, before each subsequent invocation.

- Upon successful completion, EXE$WRITECHK and EXE$WRITECHKR clear R2 to indicate a write function. EXE_STD$WRITECHK does not provide R2 as output; a driver can determine whether a function is write or read by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$WRITELOCK

Validates and prepares a user buffer for a direct-I/O, DMA read operation.

Module

SYSQIOFDT

Format

status = EXE_STD$WRITELOCK (irp, pcb, ucb, ccb, buf, bufsiz, err_rout)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required
err_rout	procedure value	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$WRITELOCK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$WRITELOCK writes the following IRP fields:

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

err_rout
Procedure value of error-handling callback routine, or 0 if the driver does not process errors.

A driver typically specifies an error-handling callback routine when it must lock multiple areas into memory for a single I/O request and must regain control to unlock these areas, if the request is to be aborted. The routine performs those tasks required before the request is backed out of or aborted. Such operations could include calling MMG_STD$UNLOCK to release previous buffers participating in the I/O operation. The error-handling routine must preserve R0 and R1 and return back to EXE_STD$WRITELOCK.

Chapter 3 describes the error-handling callback routine interface.

Return Values

SS$_NORMAL	The buffer is read-accessible and has been locked in memory.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.
SS$_INSFWSL	Insufficient working set limit.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

The system-supplied upper-level FDT action routine EXE_STD$WRITE, or a driver-specific upper-level FDT action routine, calls EXE_STD$WRITELOCK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$WRITELOCK to check the read accessibility of an I/O buffer supplied in a $QIO request for a write function, and lock the buffer in memory in preparation for a DMA write operation.

A driver cannot specify EXE_STD$WRITE for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their FDT routines to handle them.

EXE_STD$WRITELOCK invokes the $WRITECHK macro, which calls EXE_STD$WRITECHK.

EXE_STD$WRITECHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, EXE_STD$WRITECHK returns SS$_BADPARAM status to EXE_STD$READLOCK.

- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:
  - If the buffer allows read access, EXE_STD$WRITECHK returns SS$_NORMAL in R0 to EXE_STD$WRITELOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK returns SS$_ACCVIO status to EXE$_STD$READLOCK.

If error status (SS$_BADPARAM or SS$_ACCVIO) is returned, EXE_STD$WRITELOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$WRITELOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$WRITELOCK calls EXE_STD$ABORTIO, passing it the error status as **qio_sts**. EXE_STD$ABORTIO returns to EXE_STD$WRITELOCK with the error status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITELOCK immediately returns to its caller, passing these status values.

If SS$_NORMAL status is returned, EXE_STD$WRITELOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$WRITELOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$WRITELOCK. EXE_STD$WRITELOCK returns immediately to its caller, passing to it this status value.
- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$WRITELOCK. EXE_STD$WRITELOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$WRITELOCK. When

the callback routine returns (or if no callback routine is specified), EXE_STD$WRITELOCK proceeds as follows:

- For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$WRITELOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$WRITELOCK returns to its caller the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.
- For page fault status, EXE_STD$WRITELOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

The caller of EXE_STD$WRITELOCK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write accessible and has been successfully locked into memory and the starting virtual address of the page table entries that map the buffer is available in IRP$L_SVAPTE.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS. Ordinarily a driver specifies an error-handling callback routine to process such errors.

  Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$WRITELOCK.

Macro

$WRITELOCK
$WRITELOCK_ERR [interface_warning=YES]

where:

interface_warning=YES, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

In a Step 2 OpenVMS AXP driver, the $WRITELOCK simulates a JSB to EXE$WRITELOCK and $WRITELOCK_ERR simulates a JSB to EXE$WRITELOCK_ERR. $WRITELOCK calls EXE_STD$WRITELOCK, specifying 0 as the **err_rout** argument; $WRITELOCK_ERR also calls EXE_STD$WRITELOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$WRITELOCK or EXE_STD$WRITELOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and clears R2 to indicate a write operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro clears R2 to indicate a write operation and returns to FDT dispatching code in the $QIO system service.

Notes for Converting Step 1 Drivers

- EXE_STD$WRITELOCK replaces EXE$WRITELOCK and EXE$WRITELOCK_ERR (used by Step 1 OpenVMS AXP drivers). For compatibility with the Step 1 routines, use the $WRITELOCK and $WRITELOCK_ERR macros.
- EXE$WRITELOCK and EXE$WRITELOCK_ERR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$WRITELOCK.
- The order in which formal parameters are passed to EXE_STD$WRITELOCK differs from the order in which they are provided in registers to the Step 1 OpenVMS routines EXE$WRITELOCK and EXE$WRITELOCK_ERR.
- EXE$WRITELOCK_ERR provides a mechanism by which a driver callback routine or coroutine obtains control upon an error condition prior to the abortion of an I/O request. EXE_STD$WRITELOCK accepts the address of an error-handling callback routine in the **err_rout** argument. The error-handling routine is called after an I/O request encounters a buffer access or memory allocation failure and before the request is aborted.
- The design of FDT processing for Step 2 OpenVMS AXP device drivers guarantees that the caller of EXE_STD$WRITELOCK regains control whether the write lock operation is successful. When a driver regains control from a call to EXE_STD$WRITELOCK, return status in R0 indicates that the buffer has been successfully locked (SS$_NORMAL) or that the operation failed and the request has been aborted (SS$_FDT_COMPL). The driver must check the return status and take appropriate action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

  Normally, a driver services a read lock failure by supplying the address of an error-handling callback routine to EXE_STD$WRITELOCK.
- Driver code that executes after receiving failure status (SS$_FDT_COMPL) from EXE_STD$WRITELOCK cannot access information in the IRP. If the driver anticipates accessing IRP fields when EXE_STD$WRITELOCK returns, it must store these fields elsewhere before calling EXE_STD$WRITELOCK.

- Upon successful completion, EXE$WRITELOCK and EXE$WRITELOCK_ERR provide as output the system virtual address of the first process PTE that maps the buffer in R1 and in IRP$L_SVAPTE. Because EXE_STD$WRITELOCK does not provide R1 as output, a driver must obtain this information from IRP$L_SVAPTE. Similarly, the Step 1 routines clear R2 for a write function. EXE_STD$WRITELOCK does not provide R2 as output; a driver can determine whether a function is write or read by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$WRTMAILBOX

Sends a message to a mailbox.

Module

MBDRIVER

Format

status = EXE_STD$WRTMAILBOX (mb_ucb, msgsiz, msg)

Arguments

Argument	Type	Access	Mechanism	Status
mb_ucb	MB_UCB	input	reference	required
msgsiz	integer	input	value	required
msg	address	input	reference	required

mb_ucb
Mailbox UCB. (SYS$AR_JOBCTLMB contains the address of the job controller's mailbox; SYS$AR_OPRMBX contains the address of OPCOM's mailbox.)

msgsiz
Message size.

msg
Address of buffer containing the message.

Return Values

SS$_INSFMEM	The system is unable to allocate memory for the message.
SS$_MBFULL	The message mailbox is full of messages.
SS$_MBTOOSML	The message is too large for the mailbox.
SS$_NOPRIV	The caller lacks privilege to write to the mailbox.
SS$_NORMAL	Normal, successful completion.

Context

Because EXE_STD$WRTMAILBOX raises IPL to IPL$_MAILBOX and obtains the MAILBOX spin lock in a multiprocessing environment, its caller cannot be executing above IPL$_MAILBOX. EXE_STD$WRTMAILBOX returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

EXE_STD$WRTMAILBOX checks fields in the mailbox UCB (UCBW_MSGQUO, UCBW_DEVMSGSIZ) to determine whether it can deliver a message of the specified size to the mailbox. It also checks fields in the associated ORB to determine whether the caller is sufficiently privileged to write to the mailbox. Finally, it calls EXE$ALONONPAGED to allocate a block of nonpaged pool to contain the message. If it fails any of these operations, EXE_STD$WRTMAILBOX returns error status to its caller.

If it is successful thus far, EXE_STD$WRTMAILBOX creates a message and delivers it to the mailbox's message queue, adjusts its UCB fields accordingly, and returns success status to its caller.

Macro

$WRTMAILBOX [save_r1]

where:

save_r1 indicates that the macro should preserve register R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, the $WRTMAILBOX macro simulates a JSB to EXE$WRTMAILBOX in a Step 1 OpenVMS driver. $WRTMAILBOX calls EXE_STD$WRTMAILBOX, using the current contents of R5, R3, and R4 as the **mb_ucb**, **msgsiz**, and **msg** arguments, respectively. It returns status in R0. Unless you specify **save_r1=NO**, the macro preserves the R1 across the call.

Notes for Converting Step 1 Drivers

- EXE_STD$WRTMAILBOX replaces EXE$WRTMAILBOX (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). The order in which formal parameters are passed to EXE_STD$WRTMAILBOX differs from the order in which they are provided in registers to the Step 1 OpenVMS routine EXE$WRTMAILBOX.
- Unlike EXE$WRTMAILBOX, EXE_STD$WRTMAILBOX does not preserve R1 across the call.

EXE_STD$ZEROPARM

Delivers an I/O request that requires no parameters to a driver's start-I/O routine.

Module

SYSQIOFDT

Format

status = EXE_STD$ZEROPARM (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request. EXE_STD$ZEROPARM clears IRP$L_MEDIA.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$ZEROPARM as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$ZEROPARM to process an I/O function code that has no required parameters.

EXE_STD$ZEROPARM clears IRP$L_MEDIA and invokes the $QIODRVPKT macro to deliver the IRP to the driver. EXE_STD$ZEROPARM regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine EXE$ZEROPARM (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$ZEROPARM.
- EXE$ZEROPARM returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$ZEROPARM returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

IOC_STD$ALTREQCOM

Completes an I/O request for a device using the disk or tape class drivers.

Module

IOSUBNPAG

Format

IOC_STD$ALTREQCOM (iost1, iost2, cdrp, irp_p, ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
iost1	integer	input	value	required
iost2	integer	input	value	required
cdrp	CDRP	input	reference	required
irp_p	pointer	output	reference	required
ucb_p	pointer	output	reference	required

iost1
First longword of I/O status.

iost2
Second longword of I/O status.

cdrp
Class driver request packet.

irp_p
Address at which IOC_STD$ALTREQCOM writes the address of the I/O request packet.

ucb_p
Address at which IOC_STD$ALTREQCOM writes the address of the unit control block.

Context

IOC_STD$ALTREQCOM is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

$ALTREQCOM

In a Step 2 OpenVMS AXP driver, the $ALTREQCOM macro simulates a JSB to IOC$ALTREQCOM in a Step 1 OpenVMS driver. $ALTREQCOM calls IOC_STD$ALTREQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **cdrp** arguments, respectively. When IOC_STD$ALTREQCOM returns, the macro returns the address of the IRP in R3 and the address of the UCB in R4.

Notes for Converting Step 1 Drivers

- IOC_STD$ALTREQCOM replaces IOC$ALTREQCOM (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$ALTREQCOM, IOC_STD$ALTREQCOM does not return the addresses of the IRP and UCB in R3 and R5, respectively.

IOC_STD$BROADCAST

Broadcasts the specified message to a given terminal.

Module

IOSUBNPAG

Format

status = IOC_STD$BROADCAST (msglen, msg_p, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
msglen	integer	input	value	required
msg_p	address	input	reference	required
ucb	UCB	input	reference	required

msglen
Message length.

msg_p
Message.

ucb
Address of target terminal's UCB.

Return Values

SS$_ILLIOFUNC	The specified **term_ucb** is not associated with a terminal.
SS$_INSFMEM	Insufficient dynamic nonpaged pool to satisfy the request.
SS$_NORMAL	The broadcast completed successfully.

Context

IOC_STD$BROADCAST is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

$BROADCAST [save_r1]

where:

save_r1 indicates that the macro should preserve register R1 across the call to IOC_STD$BROADCAST. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, the $BROADCAST macro simulates a JSB to IOC$BROADCAST in a Step 1 OpenVMS driver. $BROADCAST calls IOC_STD$BROADCAST, using the current contents of R1, R2, and R5 as the **msglen**, **msg_p**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call.

Notes for Converting Step 1 Drivers

- IOC_STD$BROADCAST replaces IOC$BROADCAST (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$BROADCAST, IOC_STD$BROADCAST does not preserve R1 across the call.

IOC_STD$CANCELIO

Conditionally marks a UCB so that its current I/O request will be canceled.

Module

IOSUBNPAG

Format

IOC_STD$CANCELIO (chan, irp, pcb, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
chan	integer	input	value	required
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required

chan
Channel index number.

irp
I/O request packet. IOC_STD$CANCELIO reads the following IRP fields:

Field	Contents
IRP$L_PID	Process identification of the process that queued the I/O request
IRP$L_CHAN	I/O request channel index number

pcb
Current process control block.

ucb
Unit control block. IOC_STD$CANCELIO reads UCB$L_STS to determine if the device is busy (UCB$V_BSY set) or idle (UCB$V_BSY clear). IOC_STD$CANCELIO sets UCB$V_CANCEL if the I/O request should be canceled.

Context

IOC_STD$CANCELIO executes at its caller's IPL, obtains no spin locks, and returns control to its caller at the caller's IPL. It is usually called by EXE$CANCEL (if specified in the DDT as the driver's cancel-I/O routine) at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

IOC_STD$CANCELIO cancels I/O to a device in the following device-independent manner:

1. It confirms that the device is busy by examining the device-busy bit in the UCB status longword (UCB$V_BSY in UCB$L_STS).
2. It confirms that the IRP in progress on the device originates from the current process (that is, the contents of IRP$L_PID and PCB$L_PID are identical).
3. It confirms that the specified channel-index number is the same as the value stored in the IRP's channel-index field (IRP$L_CHAN).
4. It sets the cancel-I/O bit in the UCB status longword (UCB$V_CANCEL in UCB$L_STS).

Macro

$CANCELIO [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to IOC_STD$CANCELIO. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In a Step 2 OpenVMS AXP driver, the $CANCELIO macro simulates a JSB to IOC$CANCELIO in a Step 1 OpenVMS driver. $CANCELIO calls IOC_STD$CANCELIO, using the current contents of R2, R3, R4, and R5 as the **chan**, **irp**, **pcb**, and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting Step 1 Drivers

- IOC_STD$CANCELIO replaces IOC$CANCELIO (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$CANCELIO, IOC_STD$CANCELIO does not preserve R0 and R1 across the call.

IOC_STD$CLONE_UCB

Copies a template UCB and links it to the appropriate DDB list.

Module

UCBCREDEL

Format

status = IOC_STD$CLONE_UCB (tmpl_ucb, new_ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
tmpl_ucb	UCB	input	reference	required
new_ucb_p	pointer	output	reference	required

tmpl_ucb
Template unit control block.

new_ucb_p
Location into which IOC_STD$CLONE_UCB writes the address of the newly-created unit control block.

Return Values

SS$_NORMAL	UCB cloning was successful.
SS$_INSFMEM	Insufficient nonpaged pool to copy UCB.

Context

A driver calls IOC_STD$CLONE_UCB at or below IPL$_MAILBOX with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

$CLONE_UCB [interface_warning=YES]

where:

interface_warning=YES, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

In a Step 2 OpenVMS AXP driver, $CLONE_UCB simulates a JSB to IOC$CLONE_UCB. It calls IOC_STD$CLONE_UCB using the current contents of R5 as the **tmpl_ucb** argument. $CLONE_UCB returns status in R0 and the address of the newly-created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$CLONE_UCB replaces IOC$CLONE_UCB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). IOC_STD$CLONE_UCB does not return the addresses of the UCBs that precede and follow the newly-created UCB on the DDB chain.

IOC_STD$COPY_UCB

Copies and initializes a template UCB and ORB.

Module

UCBCREDEL

Format

status = IOC_STD$COPY_UCB (src_ucb, new_ucb)

Arguments

Argument	Type	Access	Mechanism	Status
src_ucb	UCB	input	reference	required
new_ucb	pointer	output	reference	required

src_ucb
Template unit control block.

new_ucb
Location into which IOC_STD$COPY_UCB writes the address of the newly-created duplicate unit control block.

Return Values

SS$_NORMAL	UCB ccpy was successful.
SS$_INSFMEM	Insufficient nonpaged pool to copy UCB.

Context

A driver calls IOC_STD$COPY_UCB at or below IPL$_MAILBOX with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

$COPY_UCB

In a Step 2 OpenVMS AXP driver, $COPY_UCB simulates a JSB to IOC$COPY_UCB. It calls IOC_STD$COPY_UCB using the current contents of R5 as the **src_ucb** argument. $CLONEUCB returns the address of the newly-created UCB in R2.

Notes for Converting Step 1 Drivers

- IOC_STD$COPY_UCB replaces IOC$COPY_UCB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). IOC_STD$COPY_UCB does not preserve the contents of R3 and R4 across the call.

IOC_STD$CREDIT_UCB

Credits the UCB charges associated with a given UCB against the process identified by the contents of UCB$L_CPID.

Module

UCBCREDEL

Format

IOC_STD$CREDIT_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Context

A driver calls IOC_STD$CREDIT_UCB at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$CREDIT_UCB

In a Step 2 OpenVMS AXP driver, $CREDIT_UCB simulates a JSB to IOC$CREDIT_UCB. It calls IOC_STD$CREDIT_UCB using the current contents of R5 as the **ucb** argument.

Notes for Converting Step 1 Drivers

- IOC_STD$CREDIT_UCB replaces IOC$CREDIT_UCB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

IOC_STD$CVT_DEVNAM

Converts a device name and unit number to a physical device name string.

Module

IOSUBNPAG

Format

status = IOC_STD$CVT_DEVNAM (buflen, buf, form, ucb, outlen_p)

Arguments

Argument	Type	Access	Mechanism	Status
buflen	integer	input	value	required
buf	address	input	reference	required
form	integer	input	value	required
ucb	UCB	reference	input	required
outlen_p	pointer	output	reference	required

buflen
Size of output buffer in bytes.

buf
Output buffer.

form
Name string formation mode, as follows:

Mode	Description
-2 (DVI$_DISPLAY_ DEVNAM)	Name suitable for displays but not suitable for $ASSIGN: "$alloclass$ddcn: (host1[, host2])", "node$ddcn", or "ddcn"
–1 (DVI$_DEVNAM)	Name suitable for displays: "node$ddcn" for non-local devices or "node$ddcn" or "ddcn" for local devices
0 (DVI$_ FULLDEVNAM)	Name with appropriate node information: either "$alloclass$ddcn" or "node$ddcn"
1 (DVI$_ ALLDEVNAM)	Name with allocation class information: either "$alloclass$ddcn" or "node$ddcn"
2 (no GETDVI item code)	Old-fashioned name: "ddcn"
3 (no GETDVI item code)	Secondary path name for displays (same as –1 except secondary path name is returned)

Mode	Description
4 (no GETDVI item code)	Path controller name for displays (same as –1 except no unit number is appended)

ucb
Unit control block for device.

outlen_p
Address of location in which IOC_STD$CVT_DEVNAM returns the length of the conversion string.

Return Values

SS$_BUFFEROVF	Successful completion, but specified buffer cannot hold the entire device name string.
SS$_NORMAL	Normal, successful completion.

Context

IOC_STD$CVT_DEVNAM is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

$CVT_DEVNAM

In a Step 2 OpenVMS AXP driver, the $CVT_DEVNAM macro simulates a JSB to IOC$CVT_DEVNAM in a Step 1 OpenVMS driver. $CVT_DEVNAM calls IOC_STD$CVT_DEVNAM, using the current contents of R0, R1, R4, and R5 as the **buflen**, **buf**, **form**, and **ucb** arguments, respectively. When IOC_STD$CVT_DEVNAM returns, the macro returns status in R0 and the length of the conversion string in R1.

Notes for Converting Step 1 Drivers

- IOC_STD$CVT_DEVNAM replaces IOC$CVT_DEVNAM (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOCCVT_DEVNAM, IOC_STDCVT_DEVNAM does not return the length of the conversion string in R1.

IOC_STD$CVTLOGPHY

Conditionally converts a logical block number to a physical disk address and stores the result in the I/O request packet.

Module

IOSUBRAMS

Format

IOC_STD$CVTLOGPHY (lbn, irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
lbn	integer	input	value	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

lbn
Logical block number to be converted.

irp
I/O request packet.

ucb
Unit control block.

Context

A driver calls IOC_STD$CVTLOGPHY at fork IPL with the corresponding fork lock held in a multiprocessing system.

Description

For Digital internal use only.

Macro

$CVTLOGPHY

In a Step 2 OpenVMS AXP driver, the $CVTLOGPHY macro simulates a JSB to IOC$CVTLOGPHY in a Step 1 OpenVMS driver. $CVTLOGPHY calls IOC_STD$CVTLOGPHY, using the current contents of R0, R3, and R5 as the **lbn**, **irp** and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$CVTLOGPHY replaces IOC$CVTLOGPHY (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$CVTLOGPHY, IOC_STD$CVTLOGPHY does not preserve R3 across the call.

IOC_STD$DELETE_UCB

Deletes the specified UCB if its reference count is zero and UCB$V_DELETEUCB is set in UCB$L_STS.

Module

UCBCREDEL

Format

IOC_STD$DELETE_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Context

A driver calls IOC_STD$DELETE_UCB with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

$DELETE_UCB

In a Step 2 OpenVMS AXP driver, $DELETE_UCB simulates a JSB to IOC$DELETE_UCB. It calls IOC_STD$DELETE_UCB using the current contents of R5 as the **ucb** argument.

Notes for Converting Step 1 Drivers

- IOC_STD$DELETE_UCB replaces IOC$DELETE_UCB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

IOC_STD$DIAGBUFILL

Fills a diagnostic buffer if the original $QIO request specified such a buffer.

Module

IOSUBNPAG

Format

IOC_STD$DIAGBUFILL (driver_param, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
driver_param	unspecified	input	reference	required
ucb	UCB	input	reference	required

driver_param

Parameter to be passed to the driver's register dumping routine. Typically, a driver supplies the address of a CRAM in this register.

ucb

Unit control block. IOC_STD$DIAGBUFILL reads the final error retry count from UCB$L_ERTCNT. It obtains the address of the current IRP from UCB$L_IRP and reads the following IRP fields:

Field	Contents
IRP$L_STS	IRP$V_DIAGBUF set if a diagnostic buffer exists
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present

IOC_STD$DIAGBUFILL obtains the address of the DDB from UCB$L_DDB and the address of the DDT from DDB$L_DDT. The procedure value of driver's register dumping routine is obtained from DDT$L_REGDUMP.

Context

The caller of IOC_STD$DIAGBUFILL may be executing at or above fork IPL and must hold the corresponding fork lock. IOC_STD$DIAGBUFILL returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

A device driver fork process calls IOC_STD$DIAGBUFILL at the end of I/O processing but before releasing the I/O channel. IOC_STD$DIAGBUFILL stores the I/O completion time and the final error retry count in the diagnostic buffer. (IOC_STD$INITIATE has already placed the I/O initiation time [from EXE$GQ_SYSTIME] in the first quadword of the buffer.) IOC_STD$DIAGBUFILL then calls the driver's register dumping routine, passing to it in the **buffer** argument an address within the diagnostic buffer in which the routine can place the register values it retrieves from device interface register space by means of hardware mailbox read transactions. It also passes the contents of the **driver_param** and **ucb** arguments. The register dumping routine fills the remainder of the buffer, and returns to IOC_STD$DIAGBUFILL, which returns to its caller.

Macro

$DIAGBUFILL

In a Step 2 OpenVMS AXP driver, the $DIAGBUFILL macro simulates a JSB to IOC$DIAGBUFILL in a Step 1 OpenVMS driver. $DIAGBUFILL calls IOC_STD$DIAGBUFILL, using the current contents of R4 and R5 as the **driver_parm** and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$DIAGBUFILL replaces IOC$DIAGBUFILL (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers).
- Prior to calling IOC_STD$DIAGBUFILL, the driver places a parameter, which the routine passes to the driver's register dumping routine, in R4. On OpenVMS AXP systems, this parameter is often the address of a CRAM (obtained, for instance, from UCB$PS_CRAM or CRB$PS_CRAM). On OpenVMS VAX systems, the parameter similarly would contain the address of the device's CSR.
- The contents of R2 and R3 are destroyed when the caller of IOC_STD$DIAGBUFILL regains control; on OpenVMS VAX systems, these registers contain the DDT address and IRP address respectively.

IOC_STD$FILSPT

Fills a system page-table entry (PTE) with the transfer PTE of a buffer that is locked in memory so that the system PTE may be directly addressed.

Module

BUFFERCTL

Format

sva = IOC_STD$FILSPT (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block. IOC_STD$FILSPT reads UCB$L_SVAPTE to obtain the system virtual address of PTE that maps the first page of the buffer.

Return Values

sva	System virtual address of the first byte in the page that contains the buffer.

Context

The caller of IOC_STD$FILSPT may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment.

Description

For Digital internal use only.

Macro

$FILSPT

In a Step 2 OpenVMS AXP driver, $FILSPT simulates a JSB to IOC$FILSPT. $FILSPT calls IOC_STD$FILSPT, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

Notes for Converting Step 1 Drivers

- IOC_STD$FILSPT replaces IOC$FILSPT (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

IOC_STD$GETBYTE

Fetches a single byte of data from a user buffer.

Module

BUFFERCTL

Format

byte = IOC_STD$GETBYTE (sva, ucb, sva_p)

Arguments

Argument	Type	Access	Mechanism	Status
sva	address	input	reference	required
ucb	UCB	input	reference	required
sva_p	pointer	output	value	required

sva
System virtual address of a single-page window into the user buffer. Prior to calling IOC_STD$GETBYTE, a driver must have caled IOC_STD$INITBUFWIND to map the system page-table entry to the user buffer.

ucb
Unit control block. IOC_STD$GETBYTE updates UCB$L_SVAPTE whenever a page boundary is crossed.

sva_p
Location in which IOC_STD$GETBYTE writes the updated system virtual address.

Return Values

byte	One byte of data (not zero-extended) returned from the user buffer.

Context

The caller of IOC_STD$GETBYTE may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment.

Description

For Digital internal use only.

Macro

$GETBYTE

In a Step 2 OpenVMS AXP driver, $GETBYTE simulates a JSB to IOC$GETBYTE. $GETBYTE calls IOC_STD$GETBYTE, passing the current contents of R0 and R5 as the **sva** and **ucb** arguments, respectively. It returns in R0 the byte of data (not zero-extended) returned from the user buffer. It returns in R1 the updated system virtual address.

Notes for Converting Step 1 Drivers

- IOC_STD$GETBYTE replaces IOC$GETBYTE (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike IOC$GETBYTE, IOC_STD$GETBYTE returns the byte of data (and not the updated system virtual address) in R0.

IOC_STD$INITBUFWIND

Initializes a single-page window into a user buffer.

Module

BUFFERCTL

Format

sva = IOC_STD$INITBUFWIND (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block. IOC_STD$INITBUFWIND initializes UCB$L_SVAPTE with the system virtual address of the page-table entry that maps the first page of the buffer.

Return Values

sva	System virtual address of the first byte in the page that contains the buffer.

Context

The caller of IOC_STD$INITBUFWIND may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment.

Description

For Digital internal use only.

Macro

$INITBUFWIND

In a Step 2 OpenVMS AXP driver, $INITBUFWIND simulates a JSB to IOC$INITBUFWIND. $INITBUFWIND calls IOC_STD$INITBUFWIND, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

Notes for Converting Step 1 Drivers

- IOC_STD$INITBUFWIND replaces IOC$INITBUFWIND (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

IOC_STD$INITIATE

Initiates the processing of the next I/O request for a device unit.

Module

IOSUBNPAG

Format

IOC_STD$INITIATE (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet. IOC_STD$INITIATE reads the following IRP fields:

Field	Contents
IRP$L_SVAPTE	Address of system buffer (buffered I/O) or system virtual address of the PTE that maps process buffer (direct I/O)
IRP$L_BOFF	Byte offset of start of buffer
IRP$L_BCNT	Size in bytes of transfer
IRP$W_STS	IRP$V_DIAGBUF set if a diagnostic buffer exists
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present. IOC_STD$INITIATE writes the current system time from EXE$GQ_SYSTIME into the first quadword of this buffer.

ucb
Unit control block. IOC_STD$INITIATE reads the following UCB fields:

Field	Contents
UCB$L_DDB	Address of DDB.
UCB$L_DDT	Address of DDT. DDT$PS_START contains the procedure value of the driver's start-I/O routine.
UCB$L_AFFINITY	Device's affinity mask.

IOC_STD$INITIATE writes the following UCB fields:

Field	Contents
UCB$L_IRP	Address of IRP
UCB$L_SVAPTE	IRP$L_SVAPTE
UCB$L_BOFF	IRP$L_BOFF
UCB$L_BCNT	IRP$L_BCNT
UCB$L_STS	UCB$V_CANCEL and UCB$V_TIMOUT cleared

Context

IOC_STD$INITIATE is called at fork IPL with the corresponding fork lock held in a multiprocessing system. Within this context, it transfers control to the driver's start-I/O routine.

Description

IOC_STD$INITIATE creates the context in which a driver fork process services an I/O request. IOC_STD$INITIATE creates this context and activates the driver's start-I/O routine in the following steps:

- Checks the CPU ID of the local processor against the device's affinity mask to determine whether the local processor can initiate the I/O operation on the device. If it cannot, IOC_STD$INITIATE takes steps to initiate the I/O function on another processor in a multiprocessing system. It then returns to its caller.
- Stores the address of the current IRP in UCB$L_IRP.
- Copies the transfer parameters contained in the IRP into the UCB:
  - Copies the address of the system buffer (buffered I/O) or the system virtual address of the PTE that maps process buffer (direct I/O) from IRP$L_SVAPTE to UCB$L_SVAPTE
  - Copies the byte offset within the page from IRP$L_BOFF to UCB$L_BOFF
  - Copies the byte count from IRP$L_BCNT to UCB$L_BCNT
- Clears the cancel-I/O and timeout bits in the UCB status longword (UCB$V_CANCEL and UCB$V_TIMOUT in UCB$L_STS).
- If the I/O request specifies a diagnostic buffer, as indicated by IRP$V_DIAGBUF in IRP$L_STS, stores the system time in the first quadword of the buffer to which IRP$L_DIAGBUF points (the $QIO system service having already allocated the buffer).
- Transfers control to the driver's start-I/O routine.

Macro

$INITIATE

In a Step 2 OpenVMS AXP driver, the $INITIATE macro simulates a JSB to IOC$INITIATE in a Step 1 driver. $INITIATE calls IOC_STD$INITIATE, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$INITIATE replaces IOC$INITIATE (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers).

IOC_STD$LINK_UCB

Searches the UCB list attached to the device data block identified by the specified UCB and links the specified UCB into the list in ascending unit number order.

Module

UCBCREDEL

Format

status = IOC_STD$LINK_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Return Values

SS$_NORMAL	Link operation was successful.
SS$_OPINCOMPL	Link operation failed due to the presence of a UCB with the same unit number as the specified UCB.

Context

A driver calls IOC_STD$LINK_UCB with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

$LINK_UCB [interface_warning=YES]

where:

interface_warning=YES, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the Step 1 version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

In a Step 2 OpenVMS AXP driver, $LINK_UCB simulates a JSB to IOC$LINK_UCB. It calls IOC_STD$LINK_UCB using the current contents of R5 as the **ucb** argument. $LINK_UCB returns status in R0 and the address of the newly created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$LINK_UCB replaces IOC$LINK_UCB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). IOC_STD$LINK_UCB does not return the addresses of the UCBs that precede and follow the newly-created UCB on the DDB chain.

IOC_STD$MAPVBLK

Maps a virtual block to a logical block using a mapping window.

Module

IOSUBRAMS

Format

status = IOC_STD$MAPVBLK (vbn, numbytes, wcb, irp, ucb, lbn_p, notmapped_p, new_ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
vbn	integer	input	value	required
numbytes	integer	input	value	required
wcb	WCB	input	reference	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required
lbn_p	pointer	output	value	required
notmapped_p	pointer	output	value	required
new_ucb_p	pointer	output	value	required

vbn
Virtual block number.

numbytes
Number of bytes to map.

wcb
Window control block.

irp
I/O request packet.

ucb
Unit control block.

lbn_p
Address at which IOC_STD$MAPVBLK writes the logical block number of the first block it maps.

notmapped_p
Address at which IOC_STD$MAPVBLK writes the number of unmapped bytes.

new_ucb_p
Address at which IOC_STD$MAPVBLK writes the address of the updated UCB.

Return Values

status	Low bit set indicates partial map with all output parameters valid, low bit clear indicates total mapping failure with only the **notmapped_p** parameter valid.

Context

IOC_STD$MAPVBLK raises IPL to IPL$_FILSYS and obtains the corresponding spin lock to perform the mapping. As a result, it cannot be called by a driver executing above IPL 8, or by a driver is executing at IPL 8 but holds the IOLOCK8 fork lock.

Description

For Digital internal use only.

Macro

$MAPVBLK

In a Step 2 OpenVMS AXP driver, the $MAPVBLK macro simulates a JSB to IOC$MAPVBLK in a Step 1 OpenVMS driver. $MAPVBLK calls IOC_STD$MAPVBLK, using the current contents of R0, R1, R2, R3, and R5 as the **vbn**, **numbytes**, **wcb**, **irp** and **ucb** arguments, respectively. It returns status in R0, the address of the logical block number of the first block mapped in R1, the number of unmapped bytes in R2, and the address of the updated UCB in R3. If the low bit of the status value in R0 is clear, signifying failure status, only the value in R2 is valid.

Notes for Converting Step 1 Drivers

- IOC_STD$MAPVBLK replaces IOC$MAPVBLK (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$MAPVBLK, IOC_STD$MAPVBLK does not preserve R3 across the call.

IOC_STD$MNTVER

Assists a driver with mount verification.

Module

IOSUBNPAG

Format

IOC_STD$MNTVER (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet, or 0. If **irp** contains the address of an IRP, EXE_STD$MNTVER inserts the IRP at the head of the pending-I/O queue in the UCB. If it contains zero, EXE_STD$MNTVER removes the IRP from the head of the pending-I/O queue and attempts to initiate I/O processing.

ucb
Unit control block.

Context

IOC_STD$MNTVER is called at fork IPL with the corresponding fork lock held in a multiprocessing system.

Description

For Digital internal use only.

Macro

$MNTVER

In a Step 2 OpenVMS AXP driver, the $MNTVER macro simulates a JSB to IOC$MNTVER in a Step 1 OpenVMS driver. $MNTVER calls IOC_STD$MNTVER, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$MNTVER replaces IOC$MNTVER (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers).

IOC_STD$MOVFRUSER, IOC_STD$MOVFRUSER2

Move data from a user buffer to an internal buffer.

Module

BUFFERCTL

Format

pointer = IOC_STD$MOVFRUSER (sysbuf, numbytes, ucb, sysbuf_p)

pointer = EXE_STD$MOVFRUSER2 (sysbuf, numbytes, ucb, sva, sysbuf_p)

Arguments

Argument	Type	Access	Mechanism	Status
sysbuf	address	input	reference	required
numbytes	integer	input	value	required
ucb	UCB	input	reference	required
sva	address	input	reference	required
sysbuf_p	pointer	output	value	required

sysbuf
Address of internal buffer.

numbytes
Number of bytes to move.

ucb
Unit control block. IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 read the following UCB fields:

Field	Contents
UCB$L_SVAPTE	System virtual address of PTE that maps the first page of the user buffer
UCB$L_SVPN	System virtual page number of SPTE allocated to driver
UCB$L_BOFF	Byte offset within the first page to start of user buffer (IOC_STD$MOVFRUSER only)

sva
System virtual address of the byte in the user buffer after the last byte moved (IOC_STD$MOVFRUSER2 only).

buffptr
System virtual address of the byte in the user buffer after the last byte moved. IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 write this field.

Return Values

pointer	System virtual address of the byte in the internal buffer after the last byte moved.

Context

The caller of IOC_STD$MOVFRUSER or IOC_STD$MOVFRUSER2 may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

A driver calls IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 to move data from a user buffer to a device that cannot itself map the user buffer to system virtual addresses (for instance, a non-DMA device).

To use either routine, the driver must have set bit DPT$V_SVP in the driver prologue table, typically by using the **flags** argument of the DPTAB macro. This causes OpenVMS to allocate a a system page-table entry (SPTE) for driver use. (See the description of the DPTAB macro in Chapter 1 for additional information.)

In order to accomplish the move, IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 first map the user buffer using the system page-table entry (SPTE) the driver allocated in a DPTAB macro invocation. If an SPTE has not been allocated to the driver, these routines cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

IOC_STD$MOVFRUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. To begin, the driver calls IOC_STD$MOVFRUSER. For each subsequent piece, the driver calls IOC_STD$MOVFRUSER2.

Macro

```
$MOVFRUSER
$MOVFRUSER2
```

In a Step 2 OpenVMS AXP driver, $MOVFRUSER and $MOVFRUSER2 simulate a JSB to IOC$MOVFRUSER and IOC$MOVFRUSER2 respectively. $MOVFRUSER calls IOC_STD$MOVFRUSER, and $MOVFRUSER2 calls IOC_STD$MOVFRUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. $MOVFRUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

Notes for Converting Step 1 Drivers

- IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 replace IOC$MOVFRUSER and IOC$MOVFRUSER2 (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike the corresponding OpenVMS VAX routines, both OpenVMS AXP routines destroy R1 across the call. The order in which formal parameters are passed to IOC_STD$MOVFRUSER2 differs from the order in which they are provided in registers to the Step 1 OpenVMS routine IOC$MOVFRUSER2.

IOC_STD$MOVTOUSER, IOC_STD$MOVTOUSER2

Move data from an internal buffer to a user buffer.

Module

BUFFERCTL

Format

pointer = IOC_STD$MOVTOUSER (sysbuf, numbytes, ucb, sysbuf_p)

pointer = IOC_STD$MOVTOUSER2 (sysbuf, numbytes, ucb, sva, sysbuf_p)

Arguments

Argument	Type	Access	Mechanism	Status
sysbuf	address	input	reference	required
numbytes	integer	input	value	required
ucb	UCB	input	reference	required
sva	address	input	reference	required
sysbuf_p	pointer	output	value	required

sysbuf
Address of internal buffer.

numbytes
Number of bytes to move.

ucb
Unit control block. IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 read the following UCB fields:

Field	Contents
UCB$L_SVAPTE	System virtual address of PTE that maps the first page of the user buffer
UCB$L_SVPN	System virtual page number of SPTE allocated to driver
UCB$L_BOFF	Byte offset within the first page to start of user buffer (IOC_STD$MOVTOUSER only)

sva
System virtual address of the byte in the user buffer after the last byte moved (IOC_STD$MOVTOUSER2 only).

buffptr
System virtual address of the byte in the user buffer after the last byte moved. IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 write this field.

Return Values

pointer	System virtual address of the byte in the internal buffer after the last byte moved.

Context

The caller of IOC_STD$MOVTOUSER or IOC_STD$MOVTOUSER2 may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

A driver calls IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 to move data from a device to a user buffer when the device itself (for instance, a non-DMA device) cannot map the user buffer to system virtual addresses.

To use either routine, the driver must have set bit DPT$V_SVP in the driver prologue table, typically by using the **flags** argument of the DPTAB macro. This causes OpenVMS to allocate a a system page-table entry (SPTE) for driver use. (See the description of the DPTAB macro in Chapter 1 for additional information.)

In order to accomplish the move, IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 first map the user buffer using the system page-table entry (SPTE) the driver allocated in a DPTAB macro invocation. If an SPTE has not been allocated to the driver, these routines cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

IOC_STD$MOVTOUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. It handles as many pages as you need. To begin, the driver calls IOC_STD$MOVTOUSER. For each subsequent buffer to move, the driver calls IOC_STD$MOVTOUSER2.

Macro

```
$MOVTOUSER
$MOVTOUSER2
```

In a Step 2 OpenVMS AXP driver, $MOVTOUSER and $MOVTOUSER2 simulate a JSB to IOC$MOVTOUSER and IOC$MOVTOUSER2 respectively. $MOVTOUSER calls IOC_STD$MOVTOUSER, and $MOVTOUSER2 calls IOC_STD$MOVTOUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. $MOVTOUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

Notes for Converting Step 1 Drivers

- IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 replace IOC$MOVTOUSER and IOC$MOVTOUSER2 (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike the corresponding OpenVMS VAX routines, both OpenVMS AXP routines destroy R1 across the call. The order in which formal parameters are passed to IOC_STD$MOVTOUSER2 differs from the order in which they are provided in registers to the Step 1 OpenVMS routine IOC$MOVTOUSER2.

IOC_STD$PARSDEVNAM

Parses a device name string, checking its syntax and extracting the node name, allocation class number, and unit number.

Module

IOSUBNPAG

Format

status = IOC_STD$PARSDEVNAM (devnamlen, devnam, flags, unit_p, scslen_p, devnamlen_p, devnam_p, flags_p)

Arguments

Argument	Type	Access	Mechanism	Status
devnamlen	integer	input	value	required
devnam	address	input	reference	required
flags	integer	input	value	required
unit_p	pointer	output	reference	required
scslen_p	pointer	output	reference	required
devnamlen_p	pointer	output	reference	required
devnam_p	pointer	output	reference	required
flags_p	pointer	output	reference	required

devnamlen
Size of the name string.

devnam
Name string.

flags
Flags.

unit_p
Address at which IOC_STD$PARSDEVNAM writes an integer representing the unit number.

scslen_p
Address at which IOC_STD$PARSDEVNAM writes an integer representing either the length of the SCS node name, the allocation class number, or the device type code.

devnamlen_p
Address at which IOC_STD$PARSDEVNAM writes an integer representing the size of the name string.

devnam_p
Address at which IOC_STD$PARSDEVNAM writes the address of the name string.

flags_p
Address at which IOC_STD$PARSDEVNAM writes an integer that contains the flags.

Return Values

SS$_IVDEVNAM	Invalid device name string.
SS$_NORMAL	Valid device name string.

Context

IOC_STD$PARSDEVNAM is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

$PARSDEVNAM

In a Step 2 OpenVMS AXP driver, the $PARSDEVNAM macro simulates a JSB to IOC$PARSDEVNAM in a Step 1 OpenVMS driver. $PARSDEVNAM calls IOC_STD$PARSDEVNAM, using the current contents of R8, R9, and R10 as the **devnamlen**, **devnam**, and **flags** arguments, respectively. respectively. When IOC_STD$PARSDEVNAM returns, the macro returns status in R0; the unit number in R2; the length of the SCS node name at the beginning of the name string, allocation class number, or device type code in R3; the size of the name string in R8, the address of the name string in R9, and the flags in R10.

Notes for Converting Step 1 Drivers

- IOC_STD$PARSDEVNAM replaces IOC$PARSDEVNAM (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$PARSDEVNAM, IOC_STD$PARSDEVNAM does not preserve the contents of R8, R9, and R10 across the call.

IOC_STD$POST_IRP

Inserts an I/O request packet in a CPU-specific I/O postprocessing queue.

Module

IOSUBNPAG

Format

IOC_STD$POST_IRP (irp)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required

irp
I/O request block.

Context

Mount verification processing calls IOC_STD$POST_IRP at or above IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$POST_IRP

In a Step 2 OpenVMS AXP driver, $POST_IRP simulates a JSB to IOC$POST_IRP. It calls IOC_STD$POST_IRP using the current contents of R3 as the **irp** argument.

Notes for Converting Step 1 Drivers

- IOC_STD$POST_IRP replaces IOC$POST_IRP (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

IOC_STD$PTETOPFN

Returns a page frame number (PFN) from a page-table entry (PTE) that has already been determined to be invalid.

Module

BUFFERCTL

Format

pfn = IOC_STD$PTETOPFN (pte)

Arguments

Argument	Type	Access	Mechanism	Status
pte	PTE	input	reference	required

pte
Quadword page-table entry.

Return Values

pfn	Page frame number (zero-extended).

Context

The caller of IOC_STD$PTETOPFN may be executing at or above IPL 0 in kernel mode.

Description

For Digital internal use only.

Macro

$PTETOPFN

In a Step 2 OpenVMS AXP driver, $PTETOPFN simulates a JSB to IOC$PTETOPFN. $PTETOPFN extracts the quadword page-table entry from R3 and passes a pointer to it as the **pte** argument to IOC_STD$PTETOPFN. It returns the page frame number in R0.

Notes for Converting Step 1 Drivers

- IOC_STD$PTETOPFN replaces IOC$PTETOPFN (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Note that, the page-table entry input argument is passed by value (in R3) to IOC$PTETOPFN, but passed by reference to IOC_STD$PTETOPFN.

IOC_STD$QNXTSEG1

Queues the next segment of a virtual I/O request that did not map to a single contiguous I/O request.

Module

IOCIOPOST

Format

IOC_STD$QNXTSEG1 (vbn, bcnt, wcb, irp, pcb, ucb, ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
vbn	integer	output	value	required
bcnt	integer	output	value	required
wcb	WCB	output	reference	required
irp	IRP	output	reference	required
pcb	PCB	output	reference	required
ucb	UCB	output	reference	required
ucb_p	pointer	input	reference	required

vbn
Virtual block number of the start of the next segment.

bcnt
Desired byte count of next segment.

wcb
Window control block.

irp
I/O request packet.

pcb
Process control block.

ucb
Unit control block.

ucb_p
Address at which IOC_STD$QNXTSEG1 writes the address of the unit control block.

Context

The caller of IOC_STD$QNXTSEG1 typically executes at or above fork IPL. IOC_STD$QNXTSEG1 executes at its caller's IPL and returns control at that IPL. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

$QNXTSEG1

In a Step 2 OpenVMS AXP driver, $QNXTSEG1 simulates a JSB to IOC$QNXTSEG1. It calls IOC_STD$QNXTSEG1 using the current contents of R0, R1, R2, R3, R4, and R5 as the **vbn**, **bcnt**, **wcb**, **irp**, **pcb**, and **ucb** arguments. It returns the address of the updated UCB in R5.

Notes for Converting Step 1 Drivers

- IOC_STD$QNXTSEG1 replaces IOC$QNXTSEG1 (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike IOC$QNXTSEG1, IOC_STD$QNXTSEG1 does not return the address of the updated UCB in R5.

IOC_STD$PRIMITIVE_REQCHANH, IOC_STD$PRIMITIVE_REQCHANL

Request a controller's data channel and, if unavailable, place process in channel wait queue.

Module

IOSUBNPAG

Format

status = IOC_STD$PRIMITIVE_REQCHANH (irp, ucb, idb_p)

status = IOC_STD$PRIMITIVE_REQCHANL (irp, ucb, idb_p)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required
idb_p	pointer	output	reference	required

irp
I/O request packet.

ucb
Unit control block. IOC_STD$PRIMITIVE_REQPCHANH and IOC_STD$PRIMITIVE_REQPCHANL read the following UCB fields:

Field	Contents
UCB$L_FPC	Procedure value of fork routine to be executed when the channel is granted if the channel cannot be granted immediately

<table>
<tr><th>Field</th><th>Contents</th></tr>
<tr><td>UCB$L_CRB</td><td>Address of controller request block (CRB). IOC_STD$REQPCHANH and IOC_STD$REQPCHANL access the following CRB fields:
<table>
<tr><th>Field</th><th>Contents</th></tr>
<tr><td>CRB$B_MASK</td><td>CRB$V_BSY set if the channel is busy</td></tr>
<tr><td>CRB$L_INTD+VEC$L_IDB</td><td>Address of IDB</td></tr>
<tr><td>CRB$L_WQFL</td><td>Head of queue of UCBs waiting for the controller channel</td></tr>
<tr><td>CRB$L_WQBL</td><td>Tail of queue of UCBs waiting for the controller channel</td></tr>
</table></td></tr>
</table>

IOC_STD$REQPCHANH and IOC_STD$REQPCHANL write the contents of the **irp** parameter in UCB$Q_FR3, and the address of the UCB in IDB$PS_OWNER.

If the channel is busy, IOC_STD$REQPCHANH and IOC_STD$REQPCHANL update CRB$L_WQFL and CRB$L_WQBL.

idb_p
Address of location in which IOC_STD$REQPCHANH and IOC_STD$REQPCHANL write the address of the interrupt dispatch block (IDB).

Return Values

SS$_NORMAL	Channel has been granted immediately.
0	Channel is busy and UCB fork block has been queued on channel-wait queue.

Context

A driver calls IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL at fork IPL holding the appropriate fork lock. Either IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL, unlike the corresponding OpenVMS VAX system routine, returns to its caller and not to its caller's caller. Each assumes that, prior to the call, its caller has placed the procedure value of the fork routine into UCB$L_FPC.

If the requested channel is busy, either IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL preserves the contents of the **irp** parameter in UCB$Q_FR3 . IOC_STD$RELCHAN eventually calls the fork routine upon granting the channel request, passing the **irp**, **idb**, and **ucb** parameters.

Description

A driver fork process calls IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL to acquire ownership of the controller's data channel.

Each routine examines CRB$V_BSY in CRB$B_MASK. If the selected controller's data channel is idle, the routine grants the channel to the fork process, placing its UCB address in IDB$PS_OWNER and returning successfully with the IDB address in the location specified by the **idb_p** parameter.

If the data channel is busy, the routine saves process context by placing the IRP address, as specified in the **irp** parameter, into the UCB fork block. IOC_STD$REQCHANH then inserts the UCB at the head of the channel wait queue (CRB$L_WQFL); IOC_STD$REQCHANL inserts the UCB at the tail of the queue (CRB$L_WQBL). Finally, the routine returns control to its caller.

When the controller channel is available to a waiting fork process, IOC_STD$RELCHAN resumes the suspended fork process at its channel grant routine, passing to it the **irp**, **idb**, and **ucb** parameters.

Macro

REQCHAN

REQPCHAN

Notes for Converting Step 1 Drivers

- IOC_STD$PRIMITIVE_REQPCHANH and IOC_STD$PRIMITIVE_REQPCHANL replace the Step 1 OpenVMS AXP routines IOC$PRIMITIVE_REQPCHANH and IOC$PRIMITIVE_REQPCHANL. The OpenVMS VAX routines IOC$REQPCHAN and IOC$REQPCHANL are not provided on OpenVMS AXP systems.

IOC_STD$PRIMITIVE_WFIKPCH, IOC_STD$PRIMITIVE_WFIRLCH

Suspend a driver fork thread and fold its context into a fork block in anticipation of a device interrupt or timeout.

Module

IOSUBNPAG

Format

IOC_STD$PRIMITIVE_WFIKPCH (irp, fr4, ucb, tmo, restore_ipl)

IOC_STD$PRIMITIVE_WFIRLCH (irp, fr4, ucb, tmo, restore_ipl)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
fr4	int64	input	value	required
ucb	UCB	input	reference	required
tmo	integer	input	value	required
restore_ipl	int	input	value	required

irp
I/O request packet.

fr4
Parameter to be passed to the interrupt service routine or timeout handling routine.

ucb
Unit control block. IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH read the following UCB fields:

Field	Contents
UCB$L_FPC	Procedure value of fork routine which may be the destination of a JSB instruction issued by either the driver's interrupt service routine or EXE$TIMEOUT
UCB$B_FLCK	Fork lock index

IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH write the following UCB fields:

Field	Contents
UCB$L_DUETIM	Sum of timeout value and EXE$GL_ABSTIM

Field	Contents
UCB$L_STS	UCB$V_INT is set to indicate that interrupts are expected on the device; UCB$V_TIM is set to indicate device I/O is being timed; and UCB$V_TIMOUT is cleared to indicate that unit has not yet timed out.
UCB$Q_FR3	R3 of caller
UCB$Q_FR4	R4 of caller

tmo
Timeout value in seconds.

restore_ipl
IPL to which to lower before returning to caller. This IPL must be the fork IPL associated with device processing and at which the driver was executing prior to invoking the DEVICELOCK macro.

Context

When it is called, IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH assumes that the local processor has obtained the appropriate synchronization with the device database by securing the appropriate device lock, as recorded in the unit control block (UCB$L_DLCK) of the device unit from which the interrupt is expected. This requirement also presumes that the local processor is executing at the device IPL associated with the lock.

Before exiting, IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH conditionally releases the device lock, so that if the caller of the driver fork thread (the caller's caller) previously owned the device lock, it will continue to hold it when it regains control. IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH also lowers the local processor's IPL to the IPL specified in the **restore_ipl** parameter.

Description

A driver fork process calls IOC_STD$PRIMITIVE_WFIKPCH to wait for an interrupt while keeping ownership of the controller's data channel; IOC_STD$PRIMITIVE_WFIRLCH, by contrast, releases the channel.

Either routine performs the following operations:

- Moves contents of the **irp** and **fr4** parameters into the UCB fork block.
- Sets UCB$V_INT to indicate an expected interrupt from the device unit.
- Sets UCB$V_TIM to indicate that OpenVMS should check for timeouts from the device unit.
- Determines the timeout due time by adding the timeout value specified in R1 to EXE$GL_ABSTIM and storing the result in UCB$L_DUETIM.
- Clears UCB$V_TIMOUT to indicate that the unit has not yet timed out.
- Invokes the DEVICEUNLOCK macro to conditionally release the device lock associated with the device unit and to lower IPL to the IPL specified in the **restore_ipl** parameter. These actions presume that the DEVICELOCK macro has been issued prior to the wait-for-interrupt invocation.

- Returns to its caller.

Note that IOC_STD$PRIMITIVE_WFIRLCH exits by transferring control to IOC_STD$RELCHAN. IOC_STD$RELCHAN releases the controller data channel and eventually issues an RSB instruction to IOC_STD$PRIMITIVE_WFIRLCH which returns to its caller. Because the release of the channel occurs at fork IPL, an interrupt service routine cannot reliably distinguish between operations initiated by IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH by examining the ownership of the CRB.

Macro

WFIKPCH

WFIRLCH

Notes for Converting Step 1 Drivers

- IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH replace the Step 1 OpenVMS AXP routines IOC$PRIMITIVE_WFIKPCH and IOC$PRIMITIVE_WFIRLCH. The OpenVMS VAX routines IOC$WFIKPCH and IOC$WFIRLCH are not provided on OpenVMS AXP systems.

IOC_STD$RELCHAN

Releases device ownership of all controller data channels.

Module

IOSUBNPAG

Format

IOC_STD$RELCHAN (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb

Unit control block. IOC_STD$RELCHAN reads UCB$L_CRB to obtain the address of the controller request block (CRB) in order to access the following CRB fields:

Field	Contents
CRB$B_MASK	CRB$V_BSY set if the channel is busy. IOC_STD$RELCHAN clears this bit if no driver is waiting for the controller channel.
CRB$L_INTD+VEC$L_IDB	Address of IDB. IOC_STD$RELCHAN obtains the address the UCB that owns the controller channel from IDB$L_OWNER. IOC_STD$RELCHAN clears IDB$L_OWNER if no driver is waiting for the controller channel.
CRB$L_WQFL	Head of queue of UCBs waiting for the controller.

Context

A driver fork process calls IOC_STD$RELCHAN at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC_STD$RELCHAN returns control to its caller after resuming execution of other fork processes waiting for a controller channel.

Description

A driver fork process calls IOC_STD$RELCHAN to release all controller data channels assigned to a device.

If the channel wait queue contains waiting fork processes, IOC_STD$RELCHAN dequeues a process, assigns the channel to that process and calls the suspended fork process at its channel grant routine, passing to it the **irp**, **idb**, and **ucb** parameters.

Macro

$RELCHAN

In a Step 2 OpenVMS AXP driver, $RELCHAN simulates a JSB to IOC$RELCHAN. It calls IOC_STD$RELCHAN using the current contents of R5 as the **ucb** argument.

Notes for Converting Step 1 Drivers

- IOC_STD$RELCHAN replaces IOC$RELCHAN (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).
- IOC$RELCHAN resumes the fork routine with the address of a device's controller and status register (CSR) in R4. Because OpenVMS AXP device drivers access device CSRs by means of a controller register access mailbox (CRAM), IOC_STD$RELCHAN provides the IDB address as input to the reactivated fork routine. The fork routine uses the IDB address as input to the driver macros and routines that manipulate CSRs by means of the CRAM.

IOC_STD$REQCOM

Completes an I/O operation on a device unit, requests I/O postprocessing of the current request, and starts the next I/O request waiting for the device.

Module

IOSUBNPAG

Format

IOC_STD$REQCOM (iost1, iost2, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
iost1	integer	input	value	required
iost2	integer	input	value	required
ucb	UCB	input	reference	required

iost1
First longword of I/O status.

iost2
Second longword of I/O status.

ucb
Unit control block. IOC_STD$REQCOM accesses the following UCB fields:

Field	Contents
UCB$L_ERTCNT	Final error count.
UCB$L_ERTMAX	Maximum error retry count.
UCB$L_EMB	Address of error message buffer.
UCB$L_IRP	Address of IRP. IOC_STD$REQCOM writes **iost1** and **iost2** into IRP$L_IOST1 and IRP$L_IOST2, respectively.
UCB$B_DEVCLASS	DC$_DISK and DC$_TAPE devices are subject to mount verification checks.
UCB$L_IOQFL	Device unit's pending-I/O queue. IOC_STD$REQCOM updates this field.

<table>
<tr><th>Field</th><th>Contents</th></tr>
<tr><td>UCB$L_STS</td><td>If error logging is in progress (that is, UCB$V_ ERLOGIP is set), IOC_STD$REQCOM writes the following fields in the error message buffer:
<table>
<tr><th>Field</th><th>Contents</th></tr>
<tr><td>EMB$L_DV_STS</td><td>UCB$L_STS.</td></tr>
<tr><td>EMB$L_DV_ERTCNT</td><td>UCB$L_ERTCNT.</td></tr>
<tr><td>EMB$L_DV_ ERTCNT+1</td><td>UCB$L_ERTMAX.</td></tr>
<tr><td>EMB$Q_DV_IOSB</td><td>Quadword of I/O status.</td></tr>
</table>
IOC_STD$REQCOM then clears UCB$V_BSY and UCB$V_ERLOGIP.</td></tr>
<tr><td>UCB$L_OPCNT</td><td>Unit operations count. IOC_STD$REQCOM increments this field.</td></tr>
</table>

Context

A driver fork process calls IOC_STD$REQCOM at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC_STD$REQCOM transfers control to IOC_STD$RELCHAN, which may call the OpenVMS fork dispatcher to resume another driver fork process. When it regains control, IOC_ STD$REQCOM returns to the driver fork process.

Description

A driver fork process calls this routine after a device I/O operation and all device-dependent processing of an I/O request is complete.

IOC$REQCOM performs the following tasks:

- If error logging is in progress for the device (as indicated by UCB$V_ ERLOGIP in UCB$L_STS), writes into the error message buffer the status of the device unit, the error retry count for the transfer, the maximum error retry count for the driver, and the final status of the I/O operation. It then releases the error message buffer by calling ERL_STD$RELEASEMB.
- Increments the device unit's operations count (UCB$L_OPCNT).
- If UCB$B_DEVCLASS specifies a disk device (DC$_DISK) or tape device (DC$_TAPE) and error status is reported, performs a set of checks to determine if mount verification is necessary. Tape end-of-file errors (SS$_ ENDOFFILE) are exempt from these checks. For a tape device with success status, checks to determine if CRC must be generated.
- Writes final I/O status (R0 and R1) into IRP$L_IOST1 and IRP$L_IOST2.
- Inserts the IRP in systemwide I/O postprocessing queue.
- Requests a software interrupt from the local processor at IPL$_IOPOST.
- Attempts to remove an IRP from the device's pending-I/O queue (at UCB$L_ IOQFL). If successful, it transfers control to IOC_STD$INITIATE to begin driver processing of this I/O request. If the queue is empty, it clears the unit busy bit (UCB$V_BSY in UCB$L_STS) to indicate that the device is idle.

- Exits by transferring control to IOC_STD$RELCHAN.

Macro

$REQCOM

In a Step 2 OpenVMS AXP driver, the $REQCOM macro simulates a JSB to IOC$REQCOM in a Step 1 OpenVMS driver. $REQCOM calls IOC_STD$REQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **ucb** arguments, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$REQCOM replaces IOC$REQCOM (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$REQCOM, IOC_STD$REQCOM does not return the addresses of the IRP and UCB in R3 and R5, respectively.
- The Step 1 REQCOM macro issues a JMP instruction to IOC$REQCOM, the effect of which is to effect a return to the caller of the driver fork process when IOC$REQCOM returns. The Step 2 REQCOM macro returns control to the driver fork process, which itself must issue the return to its caller.

IOC_STD$SEARCHDEV

Searches the I/O database for a specific physical device.

Module

IOSUBPAGD

Format

status = IOC_STD$SEARCHDEV (descr_p, ucb_p, ddb_p, sb_p)

Arguments

Argument	Type	Access	Mechanism	Status
descr_p	pointer	input	reference	required
ucb_p	pointer	output	reference	required
ddb_p	pointer	output	reference	required
sb_p	pointer	output	reference	required

descr_p
Descriptor of device logical name.

ucb_p
Address at which IOC_STD$SEARCHDEV writes the unit control block (UCB) address.

ddb_p
Address at which IOC_STD$SEARCHDEV writes the device data block (DDB) address.

sb_p
Address at which IOC_STD$SEARCHDEV writes the system block (SB) address.

Return Values

SS$_ACCVIO	Name string is not readable.
SS$_DEVALLOC	Device is allocated to another user.
SS$_DEVMOUNT	Device already mounted.
SS$_DEVOFFLINE	Device marked offline.
SS$_IVDEVNAM	Invalid device name string.
SS$_IVLOGNAM	Invalid logical name.
SS$_NODEVAVL	Device exists but is not available.
SS$_NONLOCAL	Nonlocal device.
SS$_NOPRIV	Insufficient privilege to access device.

SS$_NORMAL	Device found.
SS$_NOSUCHDEV	Device not found.
SS$_TEMPLATEDEV	Cannot allocate template device.
SS$_TOOMANYLNAM	Maximum logical name recursion limit exceeded.

Context

A driver calls IOC_STD$SEARCHDEV at IPL$_ASTDEL holding the I/O database mutex.

Description

For Digital internal use only.

Macro

$SEARCHDEV

In a Step 2 OpenVMS AXP driver, the $SEARCHDEV macro simulates a JSB to IOC$SEARCHDEV in a Step 1 OpenVMS driver. $SEARCHDEV calls IOC_STD$SEARCHDEV, using the current contents of R1 as the **descr_p** argument. When IOC_STD$SEARCHDEV returns, the macro returns returns status in R0, the UCB address in R1, the DDB address in R2, and the SB address in R3.

Notes for Converting Step 1 Drivers

- IOC_STD$SEARCHDEV replaces IOC$SEARCHDEV (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$SEARCHDEV, IOC_STD$SEARCHDEV does not provide the addresses of the UCB, DDB, and SB in R1, R2, and R3, respectively.

IOC_STD$SEARCHINT

Searches the I/O database for the specified device, using specified search rules.

Module

IOSUBNPAG

Format

status = IOC_STD$SEARCHINT (unit, scslen, devnamlen, devnam, flags, ucb_p, ddb_p, sb_p, lock_val_p)

Arguments

Argument	Type	Access	Mechanism	Status
unit	integer	input	value	required
scslen	integer	input	value	required
devnamlen	integer	input	value	required
devnam	address	input	reference	required
flags	integer	input	value	required
ucb_p	pointer	output	reference	required
ddb_p	pointer	output	reference	required
sb_p	pointer	output	reference	required
lock_val_p	pointer	output	reference	required

unit
Unit number.

scslen
Integer representing either the length of the SCS node name, the allocation class number, or the device type code.

devnamlen
Size of the name string.

devnam
Name string.

flags
Flags.

ucb_p
Address at which IOC_STD$SEARCHINT writes the UCB address.

ddb_p
Address at which IOC_STD$SEARCHINT writes the DDB address.

sb_p
Address at which IOC_STD$SEARCHINT writes the system block (SB) address.

lock_val_p
Address at which IOC_STD$SEARCHINT writes the address of the lock value block.

Return Values

SS$_DEVMOUNT	Device already mounted.
SS$_DEVOFFLINE	Device marked offline.
SS$_NODEVAVL	Device exists but is not available.
SS$_NOPRIV	Insufficient privilege to access device.
SS$_NORMAL	Device found.
SS$_NOSUCHDEV	Device not found.
SS$_TEMPLATEDEV	Cannot allocate template device.

Context

A driver calls IOC_STD$SEARCHINT at IPL$_ASTDEL holding the I/O database mutex. It may be called at elevated IPL only for searches specifying IOC$V_ANY.

Description

For Digital internal use only.

Macro

$SEARCHINT

In a Step 2 OpenVMS AXP driver, the $SEARCHINT macro simulates a JSB to IOC$SEARCHINT in a Step 1 OpenVMS driver. $SEARCHINT calls IOC_STD$SEARCHINT, using the current contents of R2, R3, R8, R9 and R10 as the **unit**, **scslen**, **devnamlen**, **devnam**, and **flags** arguments, respectively. When IOC_STD$SEARCHINT returns, the macro returns status in R0, the UCB address in R5, the DDB address in R6, and the SB address in R7.

Notes for Converting Step 1 Drivers

- IOC_STD$SEARCHINT replaces IOC$SEARCHINT (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers). Unlike IOC$SEARCHINT, IOC_STD$SEARCHINT does not provide the addresses of the UCB, DDB, and SB in R5, R6, and R7, respectively.

IOC_STD$SENSEDISK

Copies the disk's size in logical blocks from the device's UCB into the second longword of the I/O status block (IOSB) specified in a $QIO system service call, and completes the I/O operation successfully.

Module

IOSUBRAMS

Format

status = IOC_STD$SENSEDISK (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls IOC_STD$SENSEDISK as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine IOC$SENSEDISK (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to IOC_STD$SENSEDISK.
- IOC_STD$SENSEDISK returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. IOC_STD$SENSEDISK returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

IOC_STD$SEVER_UCB

Removes the specified UCB from the UCB list of the device data block identified within the specified UCB.

Module

UCBCREDEL

Format

IOC_STD$SEVER_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Context

A driver calls IOC_STD$SEVER_UCB with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

$SEVER_UCB

In a Step 2 OpenVMS AXP driver, $SEVER_UCB simulates a JSB to IOC$SEVER_UCB. It calls IOC_STD$SEVER_UCB using the current contents of R5 as the **ucb** argument.

Notes for Converting Step 1 Drivers

- IOC_STD$SEVER_UCB replaces IOC$SEVER_UCB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

IOC_STD$SIMREQCOM

Completes an I/O operation by setting an event flag, modifying an I/O status block (IOSB), setting an event flag, or queuing an AST to the process requesting the I/O. The caller of this routine is responsible for checking quotas and updating the I/O count.

Module

IOCIOPOST

Format

status = IOC_STD$SIMREQCOM (iosb, pri, efn, iost, acb, acmode)

Arguments

Argument	Type	Access	Mechanism	Status
iosb	IOSB	input	reference	optional
pri	integer	input	value	optional
efn	integer	input	value	optional
iost	unspecified	input	unspecified	required
acb	ACB	input	reference	optional
acmode	integer	input	value	optional

iosb

I/O status block. If this parameter contains the address of an IOSB, IOC_STD$SIMREQCOM checks for write access to the IOSB. If it contains a zero, IOC_STD$SIMREQCOM makes no IOSB modifications.

pri

Priority boost class to be passed directly to SCH$POSTEF and SCH$QAST. If an IOSB address is supplied to the **iosb** parameter, this parameter has no effect. If this parameter contains a zero, there is no priority boost.

efn

Common or local event flag to be set. If this parameter contains –1, no event flag is set.

iost

Internal process identification (IPID) of the target process (if the **iosb** parameter is zero); address of a quadword containing the new contents of the user's IOSB (if the **iosb** is non-zero).

acb

AST control block. If this parameter is zero, no AST is delivered. When the **acb** parameter is non-zero and ACB$L_AST is zero, IOC_STD$SIMREQCOM checks ACB$V_NODELETE. If ACB$V_NODELETE is clear, IOC_STD$SIMREQCOM uses ACB$W_SIZE to return the ACB and any structure in which it is embedded to nonpaged pool.

acmode
Access mode of the process originally requesting the I/O operation. IOC_STD$SIMREQCOM uses this value to probe the IOSB (if specified) for write access. If the **iosb** parameter is zero, this parameter is ignored.

Return Values

SS$_ILLEFC	Illegal cluster number.
SS$_NONEXPR	Nonexistent process.
SS$_NORMAL	Normal, successful completion.
SS$_UNASEFC	Unassigned cluster number.
SS$_WASCLR	Specified event flag was clear initially.
SS$_WASSET	Specified event flag was set initially.

Context

If supplying a non-zero value for the **iosb** parameter, the caller of IOC_STD$SIMREQCOM must be executing in the context of the target process.

Description

For Digital internal use only.

Macro

$SIMREQCOM

In a Step 2 OpenVMS AXP driver, the $SIMREQCOM macro simulates a JSB to IOC$SIMREQCOM in a Step 1 OpenVMS driver. $SIMREQCOM calls IOC_STD$SIMREQCOM, using the current contents of R1, R2, R3, R4, R5, and R6 as the **iosb**, **pri**, **efn**, **iost**, **acb**, and **acmode** arguments, respectively.

Notes for Converting Step 1 Drivers

- IOC_STD$SIMREQCOM replaces IOC$SIMREQCOM (used by Step 1 OpenVMS AXP and OpenVMS VAX drivers).

IOC_STD$THREADCRB

Threads a controller request block (CRB) onto the CRB timeout queue chain headed by IOC$GL_CRBTMOUT.

Module

IOSUBNPAG

Format

IOC_STD$THREADCRB (crb)

Arguments

Argument	Type	Access	Mechanism	Status
crb	CRB	input	reference	required

crb
Controller request block.

Context

Mount verification processing calls IOC_STD$THREADCRB at or above IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$THREADCRB [save_r0]

where:

save_r0 indicates that the macro should preserve register R0 across the call to IOC_STD$THREADCRB. If **save_r0** is blank or **save_r0=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r0=NO**, R0 is not saved.)

In a Step 2 OpenVMS AXP driver, $THREADCRB simulates a JSB to IOC$THREADCRB. It calls IOC_STD$THREADCRB using the current contents of R3 as the **crb** argument. Unless you specify **save_r1=NO**, the macro preserves the quadword register R1 across the call.

Notes for Converting Step 1 Drivers

- IOC_STD$THREADCRB replaces IOC$THREADCRB (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike IOC$THREADCRB, IOC_STD$THREADCRB routines does not preserve R0 across the call.

MMG_STD$IOLOCK

Locks process pages in memory.

Module

IOLOCK

Format

status = MMG_STD$IOLOCK (buf, bufsize, is_read, pcb, svapte_p)

Arguments

Argument	Type	Access	Mechanism	Status
buf	address	input	reference	required
bufsize	integer	input	value	required
is_read	integer	input	value	required
pcb	PCB	input	reference	required
svapte_p	pointer	output	reference	required

buf
Buffer.

bufsize
Size of output buffer in bytes.

is_read
Transfer direction indicator, as follows:

Value	Description
0	Write from memory to I/O device
1	Read into memory from I/O device
5	Write from and read into memory from I/O device

pcb
Process control block.

svapte_p
Address of location in which MMG_STD$IOLOCK returns either the system virtual address of the first page-table entry (if the returned status is SS$_NORMAL) or the address of a page to be faulted into memory (if the returned status is 0).

Return Values

SS$_ACCVIO	Specified buffer is not a process buffer, but does not fully reside in system space; or process buffer overruns balance set slots.
SS$_INSFWSL	Insufficient working set list.
SS$_NORMAL	Normal, successful completion.
0	Virtual address must be faulted into memory.

Context

MMG_STD$IOLOCK must be called at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$IOLOCK

In a Step 2 OpenVMS AXP driver, $IOLOCK simulates a JSB to MMG$IOLOCK. It calls MMG_STD$IOLOCK using the current contents of R0, R1, R2, and R4 as the **buf**, **bufsize**, **is_read**, and **pcb** arguments, respectively.

$IOLOCK returns status in R0. If R0 contains SS$_NORMAL, R1 contains the system virtual address of the first page-table entry. If R0 contains zero, R1 contains the address of a page to be faulted into memory. R0 can also contain a system-level status.

Notes for Converting Step 1 Drivers

- MMG_STD$IOLOCK replaces MMG$IOLOCK (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

MMG_STD$UNLOCK

Unlocks process pages previously locked for a direct-I/O operation.

Module

IOLOCK

Format

MMG_STD$UNLOCK (npages, svapte)

Arguments

Argument	Type	Access	Mechanism	Status
npages	integer	input	value	required
svapte	integer	input	value	required

npages
Number of buffer pages to unlock.

svapte
System virtual address of PTE for the first buffer page.

Context

Because MMG_STD$UNLOCK raises IPL to IPL$_SYNCH, and obtains the MMG spin lock in a multiprocessing environment, its caller cannot be executing above IPL$_SYNCH or hold any higher ranked spin locks. MMG_STD$UNLOCK returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

Drivers rarely use MMG_STD$UNLOCK. At the completion of a direct-I/O transfer, IOC_STD$IOPOST automatically unlocks the pages of both the user buffer and any additional buffers specified in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the packet undergoing completion processing.

However, driver FDT routines do use MMG_STD$UNLOCK when an attempt to lock IRPE buffers for a direct-I/O transfer fails. The buffer-locking routines called by such a driver—EXE_STD$READLOCK, EXE_STD$WRITELOCK, and EXE_STD$MODIFYLOCK – all allow a driver to specify an error handling callback routine that can callMMG_STD$UNLOCK to unlock all previously locked regions and deallocate the IRPE (using EXE_STD$DEANONPAGED).

Macro

$UNLOCK

In a Step 2 OpenVMS AXP driver, $UNLOCK simulates a JSB to MMG$UNLOCK. It calls MMG_STD$UNLOCK using the current contents of R1 and R3 as the **npages** and **svapte** arguments, respectively.

Notes for Converting Step 1 Drivers

- MMG_STD$UNLOCK replaces MMG$UNLOCK (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers).

MT_STD$CHECK_ACCESS

Checks access rights for magtape control write functions.

Module

MTFDT

Format

status = MT_STD$CHECK_ACCESS (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Process does not have write access to volume.
SS$_NORMAL	I/O request has been successfully queued to the driver's start-I/O routine.
SS$_NOPRIV	Process has insufficient privileges to perform a control write function.
SS$_WRITLCK	Device software is write locked.

Context

FDT dispatching code in the $QIO system service calls MT_STD$CHECK_ACCESS as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting Step 1 Drivers

- The FDT routine MT$CHECK_ACCESS (used by OpenVMS VAX and Step 1 OpenVMS AXP device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to MT_STD$CHECK_ACCESS.
- Upon a successful return from MT$CHECK_ACCESS, its OpenVMS VAX and Step 1 OpenVMS AXP callers needed to call EXE$ZEROPARM to queue the request to the driver's start-I/O routine.

  If the volume is not write-locked and the requesting process has write access to the volume. MT_STD$CHECK_ACCESS automatically invokes the $QIODRVPKT macro.
- MT$CHECK_ACCESS returns control to the system service dispatcher, passing it the final $QIO system service status in R0. MT_STD$CHECK_ACCESS returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

SCH_STD$IOLOCKR

Locks the I/O database mutex on behalf of its caller for read access.

Module

MUTEX

Format

pointer = SCH_STD$IOLOCKR (pcb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required

pcb
Process control block.

Return Values

pointer	Address of I/O database mutex.

Context

SCH_STD$IOLOCKR must be called at or below IPL$_SYNCH. It returns to its caller at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$IOLOCKR [save_r1]

where:

save_r1 indicates that the macro should preserve register R1 across the call to SCH_STD$IOLOCKR. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, $IOLOCKR simulates a JSB to SCH$IOLOCKR. It calls SCH_STD$IOLOCKR using the current contents of R4 as the **pcb** argument.

$IOLOCKR returns the address of the I/O database mutex in R0. Unless you specify **save_r1=NO**, the macro preserves R1 across the call.

Notes for Converting Step 1 Drivers

- SCH_STD$IOLOCKR replaces SCH$IOLOCKR (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike SCH$IOLOCKR, SCH_STD$IOLOCKR destroys the contents of R1 through R3 across the call.

SCH_STD$IOLOCKW

Locks the I/O database mutex on behalf of its caller for write access.

Module

MUTEX

Format

pointer = SCH_STD$IOLOCKW (pcb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required

pcb
Process control block.

Return Values

pointer	Address of I/O database mutex.

Context

SCH_STD$IOLOCKW must be called at or below IPL$_SYNCH. It returns to its caller at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

$IOLOCKW [save_r1]

where:

save_r1 indicates that the macro should preserve register R1 across the call to SCH_STD$IOLOCKW. If **save_r1** is blank or **save_r1=YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1=NO**, R1 is not saved.)

In a Step 2 OpenVMS AXP driver, $IOLOCKW simulates a JSB to SCH$IOLOCKW. It calls SCH_STD$IOLOCKW using the current contents of R4 as the **pcb** argument.

$IOLOCKW returns the address of the I/O database mutex in R0. Unless you specify **save_r1=NO**, the macro preserves R1 across the call.

Notes for Converting Step 1 Drivers

- SCH_STD$IOLOCKW replaces SCH$IOLOCKW (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike SCH$IOLOCKW, SCH_STD$IOLOCKW destroys the contents of R1 through R3 across the call.

SCH_STD$IOUNLOCK

Releases ownership of the I/O database mutex and, if the mutex has thus become available to waiting processes, reactivates the next eligible process.

Module

MUTEX

Format

SCH_STD$IOUNLOCK (pcb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required

pcb
Process control block.

Context

SCH_STD$IOUNLOCK must be called below IPL$_SCHED. It returns to its caller at its caller's IPL.

Description

For Digital internal use only.

Macro

$IOUNLOCK

In a Step 2 OpenVMS AXP driver, $IOUNLOCK simulates a JSB to SCH$IOUNLOCK. It calls SCH_STD$IOUNLOCK using the current contents of R4 as the **pcb** argument.

Notes for Converting Step 1 Drivers

- SCH_STD$IOUNLOCK replaces SCH$IOUNLOCK (used by Step 1 OpenVMS AXP drivers and OpenVMS VAX drivers). Unlike SCH$IOUNLOCK, SCH_STD$IOUNLOCK destroys the contents of R1 through R3 across the call.

3

Device Driver Entry Points

This chapter describes the standard driver routines that OpenVMS AXP uses as entry points in a device driver program. These include:

- Alternate start-I/O routine
- Cancel-I/O routine
- Cancel selective routine[1]
- Channel assign routine[1]
- Cloned unit control block (UCB)
- Controller initialization
- FDT
- Interrupt service
- Mount verification routine[1]
- Register dumping
- Start-I/O (both traditional and kernel process)
- Timeout handling
- Unit delivery (both traditional and kernel process)
- Unit initialization

Field Test 1 Note

For Field Test 1, this section does not include the following routine descriptions. They will be included in the next FT2 draft.

- Driver Start I/O Routine (JSB)
- Fork Routine Interface
- Driver Channel Grant Fork Routine Entry
- Driver Interrupt Service Routine Entry
- Driver Device Timeout Routine Entry
- Driver Resume From Interrupt Routine Entry

Note that, unlike OpenVMS VAX, OpenVMS AXP does not support driver unloading routines and unsolicited interrupt handling routines.

[1] Reserved to Digital.

This chapter also describes the Step 2 driver-entry-point macros that replace the .JSB_ENTRY directive used in the Step 1 OpenVMS AXP driver entry points. In general, these macros perform the following operations:

1. Declare a call entry point.
2. Specify a register save list that consists of the registers that the step 1 JSB interface was allowed to scratch. This save list augments the list of auto-preserved registers detected by the MACRO-32 compiler. You can specify an alternative save list if you are certain that the default mask contains registers that are not used in the execution path initiated by the entry point.
3. Define symbolic AP offsets that correspond to the routine parameters.
4. Copy the input parameters into the registers that correspond to the input registers of the Step 1 JSB interface. This register loading can be disabled by an optional parameter.

For more information about optimizing the use of the driver-entry-point macros, see *OpenVMS Step 2 Device Driver Reference Manual*.

Alternate Start-I/O Routine

Initiates activity on a device that can support multiple, concurrent I/O operations and synchronizes access to its UCB.

Format

ALTSTART (irp, ucb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
ucb	UCB	input	reference

irp
I/O request packet for the current I/O request

ucb
Unit control block of the device that is the target of the I/O request

Essentials

Identifying the Routine
Specify the address of the alternate start-I/O routine in the **altstart** argument to the DDTAB macro. This macro places the procedure value of the routine into the DDT.

Declaring the Entry Point

Use:

$DRIVER_ALTSTART_ENTRY [preserve=<R2,R3,R4,R5>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the alternate start-I/O routine
fetch=YES, the default, loads the addresses of the IRP and UCB into R3 and R5, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver alternate start-I/O routine that uses this macro can access any of its arguments by using a symbolic name of the form ALTARG$**_argument-name**.

♦

Called by
Called by routine EXE_STD$ALTQUEPKT in module SYSQIOREQ. A driver FDT routine typically is the caller of EXE_STD$ALTQUEPKT.

Step 1 OpenVMS AXP Device Driver Entry Points
Alternate Start-I/O Routine

Context
An alternate start-I/O routine begins execution at fork IPL, holding the corresponding fork lock. It must return control to EXE_STD$ALTQUEPKT in this context.

Because an alternate start-I/O routine gains control in fork process context, it can access only those virtual addresses that are in system (S0) space.

Exit mechanism
The alternate start-I/O routine completes I/O requests by calling COM_STD$POST. This routine places each IRP in the I/O postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. If no IRPs remain, the driver returns control to EXE_STD$ALTQUEPKT, which relinquishes fork level synchronization and returns to the driver FDT routine that called it. The FDT routine performs any required postprocessing and calls the FDT completion routine EXE_STD$QIORETURN.

Description

An alternate start-I/O routine initiates requests for activity on a device that can process two or more I/O requests simultaneously. Because the method by which the alternate start-I/O routine is invoked bypasses the unit's pending-I/O queue (UCB$L_IOQFL) and the device busy flag (UCB$V_BSY in UCB$L_STS), the routine is activated regardless of whether the device unit is busy with another request.

As a result, the driver that incorporates an alternate start-I/O routine must use its own internal I/O queues (in a UCB extension, for instance) and maintain synchronization with the unit's pending-I/O queue. In addition, if the routine processes more than one IRP at a time, it must use separate fork blocks for each request.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of the routine with a $DRIVER_ALTSTART_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- You should examine the routine's use of suspension mechanisms (for instance, its forking, wait-for-interrupt, and resource-wait semantics) to determine whether it needs to be adapted to use the kernel process services. Typically a driver that makes subroutine calls before suspending itself (and relies on the previous context of these subroutines remaining intact on the stack), must be adapted to use the kernel process services.
- If the routine need not be converted to a kernel process, you should replace any calls to EXE$FORK, EXE$FORK_WAIT, EXE$IOFORK, IOC$WFIKPCH, IOC$WFIRLCH, IOC$REQPCHANH, and IOC$REQPCHANL with invocations of the appropriate suspension macro or with calls to EXE_STD$PRIMITIVE_FORK, IOC_STD$PRIMITIVE_WFIKPCH, IOC_

STD$PRIMITIVE_WFIRLCH, IOC_STD$PRIMITIVE_REQCHANH, or IOC_STD$PRIMITIVE_REQCHANL.

Cancel-I/O Routine

Prevents further device-specific processing of the I/O request currently being processed on a device.

Format

CANCEL (chan, irp, pcb, ucb, reason)

Arguments

Argument	Type	Access	Mechanism
chan	integer	input	value
irp	IRP	input	reference
pcb	PCB	input	reference
ucb	UCB	input	reference
reason	integer	input	value

chan
Channel index number.

irp
I/O request packet, if any, for device (contents of UCB$L_IRP).

pcb
Process control block of process for which the I/O request is being canceled.

ucb
Unit control block.

reason
Reason for cancellation, one of the following:

CAN$C_CANCEL	Called by $CANCEL system service
CAN$C_DASSGN	Called by $DASSGN or $DALLOC system service

Essentials

Identifying the Routine

Supply the address of the cancel-I/O routine in the **cancel** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT. Many drivers specify the system routine IOC_STD$CANCELIO as their cancel-I/O routine.

Declaring the Entry Point

Use:

$DRIVER_CANCEL_ENTRY [preserve=<R2,R3,R4>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel I/O routine

fetch=YES, the default, loads the channel index number into R2, the cancellation reason into R8, and the addresses of the IRP, PCB, and UCB into R3, R4, R5, and R6, respectively. **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver cancel-I/O routine that uses this macro can access any of its arguments by using a symbolic name of the form CANARG$**_argument-name**.

♦

Called by

System routines call a driver's cancel-I/O routine under the following circumstances:

- When a process issues a Cancel-I/O-on-Channel system service ($CANCEL)
- When a process deallocates a device, causing the device's reference count (UCB$L_REFC) to become zero (that is, no process I/O channels are assigned to the device)
- When a process deassigns a channel from a device, using the $DASSGN system service
- When the command interpreter performs cleanup operations as part of image termination by canceling all pending I/O requests for the image and closing all image-related files open on process I/O channels

Context

A cancel-I/O routine begins execution at fork IPL, holding the corresponding fork lock. It must return control to its caller in this context.

A cancel-I/O routine executes in kernel mode in the context of the caller of the $CANCEL, $DALLOC, or $DASSGN system service.

Exit mechanism

The cancel-I/O routine returns to its caller.

Description

A driver's cancel-I/O routine must perform the following tasks:

1. Confirm that the device is busy by examining the device-busy bit in the UCB status longword (UCB$V_BSY in UCB$L_STS).
2. Confirm that the process ID (PID) of the request the device is servicing (IRP$L_PID) matches that of the process requesting the cancellation (PCB$L_PID).
3. Confirm that the channel-index number of the request the device is servicing (IRP$L_CHAN) matches that specified in the cancel-I/O request.

4. Cause to be completed (canceled) as quickly as possible all active I/O requests on the specified channel that were made by the process that has requested the cancellation. The cancel-I/O routine usually accomplishes this by setting UCB$V_CANCEL in the UCB$L_STS. When the next interrupt or timeout occurs for the device, the driver's start-I/O routine detects the presence of an active but canceled I/O request by testing this bit and takes appropriate action, such as completing the request without initiating any further device activity. Other driver routines, such as the timeout handling routine, check the cancel-I/O bit to determine whether to retry the I/O operation or abort it.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of a cancel-I/O routine with a $DRIVER_CANCEL_ENTRY macro, indicating which registers must be saved and restored across routine execution.

♦

Cancel Selective Routine

Performs additional processing on a list of I/O requests which have been canceled.

Format

status=CANCEL_SELECTIVE (pcb, ucb, chan, iosb_vector, iosb_count)

Arguments

Argument	Type	Access	Mechanism
pcb	PCB	input	reference
ucb	UCB	input	reference
chan	integer	input	value
iosb_vector	address	input	value
iosb_count	integer	input	value

pcb
Process control block of process for which the I/O request is being canceled.

ucb
Unit control block.

chan
Channel index number.

iosb_vector
Vector of address of I/O status blocks (IOSBs), or zero.

iosb_count
Number of addresses in the IOSB vector.

Essentials

Identifying the Routine

Supply the address of the cancel selective routine in the **cancel_selective** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT.

Declaring the Entry Point

Use:

$DRIVER_CANCEL_SELECTIVE_ENTRY [preserve=<>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel selective routine

fetch=YES, the default, loads SS$_UNSUPPORTED status into R0, the IOSB vector into R7, the IOSB count into R8, and the addresses of the PCB and UCB into R4 and R5, respectively, **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver cancel selective routine that uses this macro can access any of its arguments by using a symbolic name of the form CANSARG$**_argument-name**.

♦

Called by

EXE$CANCEL_SELECTIVE calls a driver’s cancel selective routine.

Context

A cancel selective routine is called at device IPL, holding the corresponding device lock and the appropriate fork lock. The channel control block (CCB) is locked in memory. It must return control to EXE$CANCEL_SELECTIVE in this context.

Exit mechanism

The cancel selective routine returns to its caller.

Description

Reserved to Digital.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of a cancel-I/O routine with a $DRIVER_ CANCEL_SELECTIVE_ENTRY macro, indicating which registers must be saved and restored across routine execution.

♦

Channel Assign Routine

Performs specialized operations when a channel is assigned to a non-network device.

Format

CHANNEL_ASSIGN (ucb, ccb)

Arguments

Argument	Type	Access	Mechanism
ucb	UCB	input	reference
ccb	CCB	input	reference

ucb
Unit control block.

ccb
Channel control block.

Essentials

Identifying the Routine
Supply the address of the channel assign routine in the **channel_assign** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT.

Declaring the Entry Point

Use:

$DRIVER_CHANNEL_ASSIGN_ENTRY [preserve=<>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel selective routine
fetch=YES, the default, loads the addresses of UCB and CCB into R5 and R8, respectively, **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver channel assign routine that uses this macro can access any of its arguments by using a symbolic name of the form CHANARG$**_argument-name**.

♦

Called by
EXE$ASSIGN_LOCAL (in module SYSASSIGN) calls a driver's channel assign routine.

Step 1 OpenVMS AXP Device Driver Entry Points
Channel Assign Routine

Context
A channel assign routine is called in kernel mode at IPL 0.

Exit mechanism
The channel assign routine returns to its caller.

Description

Reserved to Digital.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of a channel assign routine with a $DRIVER_CHANNEL_ASSIGN_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

Cloned UCB Routine

Completes the initialization of the UCB cloned when a channel was requested for a template device.

Format

status = CLONEDUCB (cloned_ucb, ddt, pcb, template_ucb)

Arguments

Argument	Type	Access	Mechanism
cloned_ucb	UCB	input	reference
ddt	DDT	input	reference
pcb	PCB	input	reference
template_ucb	UCB	input	reference

cloned_ucb
Cloned unit control block. Fields of the cloned UCB have been initialized as follows:

Field	Value
UCB$L_FQFL	Address of UCB$L_FQFL
UCB$L_FQBL	Address of UCB$L_FQFL
UCB$L_FPC	0
UCB$Q_FR3	0
UCB$Q_FR4	0
UCB$W_BUFQUO	0
UCB$L_LINK	Address of next UCB in DDB chain
UCB$L_IOQFL	Address of UCB$L_IOQFL
UCB$L_IOQBL	Address of UCB$L_IOQFL
UCB$W_UNIT	Device unit number
UCB$W_CHARGE	Mailbox byte quota charge (UCB$W_SIZE)
UCB$L_REFC	0
UCB$L_STS	UCB$V_DELETEUCB set, UCB$V_ONLINE set
UCB$L_DEVSTS	UCB$V_DELMBX set if DEV$V_MBX is set in UCB$L_DEVCHAR

Field	Value
UCB$L_OPCNT	0
UCB$L_SVAPTE	0
UCB$L_BOFF	0
UCB$L_BCNT	0
UCB$L_ORB	Address of object rights block (ORB) for the cloned UCB

The following fields of the template UCB's ORB are also initialized:

Field	Value
ORB$L_OWNER	UIC of current process
ORB$L_ACL_MUTEX	$FFFF_{16}$
ORB$B_FLAGS	ORB$V_PROT_16 set
ORB$W_PROT	0
ORB$L_ACL_COUNT	0
ORB$L_ACL_DESC	0
ORB$R_MIN_CLASS	0 in first longword

pcb
Process control block of the current process.

ddt
Driver dispatch table.

template_ucb
Template unit control block.

Essentials

Identifying the Routine
Specify the address of a cloned UCB routine in the **cloneducb** argument of the DDTAB macro. The macro places the procedure value of the routine into the DDT. Only drivers for template devices, such as mailboxes, specify a cloned UCB routine.

Declaring the Entry Point
Use:

```
$DRIVER_CLONEDUCB_ENTRY [preserve=<R3>] [,fetch=YES]
```

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cloned UCB routine
fetch=YES, the default, loads SS$_NORMAL status into R0, and the addresses of the cloned UCB, DDT, PCB, and template UCB into R2, R3, R4, and R5, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver cloned UCB routine that uses this

macro can access any of its arguments by using a symbolic name of the form CLONEARG$**_argument-name**.

♦

Called by

EXE$ASSIGN calls the driver’s cloned UCB routine when an Assign I/O Channel system service request ($ASSIGN) specifies a template device (that is, bit UCB$V_TEMPLATE in UCB$L_STS is set).

Context

A cloned UCB routine executes at IPL$_ASTDEL, holding the I/O database mutex (IOC$GL_MUTEX).

A cloned UCB routine executes in kernel mode in the context of the process that called the $ASSIGN system service.

Exit mechanism

A cloned UCB routine must return control and status to EXE$ASSIGN. If the routine returns error status in R0, EXE$ASSIGN undoes the process of UCB cloning and completes with failure status in R0.

Description

When a process requests that a channel be assigned to a template device, EXE$ASSIGN does not assign the channel to the template device itself. Rather, it creates a copy of the template device’s UCB and ORB, initializing and clearing certain fields as appropriate.

The driver’s cloned UCB routine verifies the contents of these fields and completes their initialization.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of a cloned UCB with a $DRIVER_CLONEDUCB_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

Controller Initialization Routine

Prepares a controller for operation.

Format

status = CTRLINIT (idb, ddb, crb)

Arguments

Argument	Type	Access	Mechanism
idb	IDB	input	reference
ddb	DDB	input	reference
crb	CRB	input	reference

idb
Interrupt dispatch block associated with the controller.

ddb
Device data block associated with the controller.

crb
Controller request block.

Essentials

Identifying the Routine

Specify the address of a controller initialization routine in the **ctrlinit** argument of the DDTAB macro. The macro places the procedure value of this routine into the DDT.

Declaring the Entry Point

Use:

$DRIVER_CTRLINIT_ENTRY [preserve=<R2>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the controller initialization routine
fetch=YES, the default, loads $SS_NORMAL status into R0, the address of the IDB into R4 and R5, and the addresses of the DDB and CRB into R6 and R8, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver controller initialization routine that uses this macro can access any of its arguments by using a symbolic name of the form CTRLARG$**_argument-name**.

♦

Called by

The driver-loading procedure calls a driver's controller initialization routine when processing a CONNECT command. Also, the system calls this routine if the device, controller, processor, or adapter to which the device is connected experiences a power failure.

Context

OpenVMS calls a controller initialization routine at IPL$_POWER. If it must lower IPL, the controller initialization routine cannot explicitly do so. Rather, it must fork. Because the driver-loading procedure calls the unit initialization routine immediately after the controller initialization returns control to it, the driver's initialization routines must synchronize their activities. If the controller initialization routine forks, the unit initialization routine must be prepared to execute before the controller initialization routine completes.

The portion of the controller initialization that services power failure cannot acquire any spin locks. As a result, the routine cannot fork to perform power failure servicing.

Because a controller initialization routine executes within system context, it can refer only to those virtual addresses that reside in system (S0) space.

Exit mechanism

The controller initialization routine returns success or failure status to its caller.

Description

Some controllers require initialization when the system's driver-loading routine loads the driver and when the system is recovering from a power failure. Depending on the device, a controller initialization routine performs any and all of the following actions:

- Determine whether it is being called as a result of a power failure by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB. A controller initialization routine may want to perform or avoid specific tasks when servicing a power failure.
- Clear error-status bits in device registers.
- Enable controller interrupts.
- Allocate resources that must be permanently allocated to the controller.
- If the controller is dedicated to a single-unit device, such as a printer, fill in IDB$PS_OWNER and set the online bit (UCB$V_ONLINE in UCB$L_STS).
- Initialize the interrupt vectors of devices with programmable interrupt vectors.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of the routine with a $DRIVER_CTRLINIT_ ENTRY macro to indicate which registers must be saved and restored across routine execution.

  ♦

- An OpenVMS VAX device driver specifies a controller initialization routine by invoking the DPT_STORE macro to place its procedure value into the interrupt transfer vector block (CRB$L_INTD+VEC$L_INITIAL). An OpenVMS AXP device driver specifies the routine in the **ctrlinit** argument of the DDTAB macro.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- The controller initialization routine of an OpenVMS VAX device driver receives the addresses of the device CSR in R4 and the IDB in R5. An OpenVMS AXP device driver's controller initialization routine is not passed the address of the CSR. It may access the controller's register by means of the controller register access mailbox (CRAM), the address of which is provided in IDB$PS_CRAM.
- A controller initialization routine that must initialize the programmable interrupt vectors of a device does so by referring to the vector offset placed in IDB$L_VECTOR by the driver-loading procedure. In the case of a device with multiple interrupt vectors, IDB$L_VECTOR contains the address of a vector list extension (VLE) which contains a list of vector offsets.
- An OpenVMS AXP controller initialization routine must return success or failure status to its caller.

Driver Unloading Routine

******Not supported in OpenVMS AXP drivers********

FDT Upper-Level Action Routine

Performs any device-dependent activities needed to prepare the I/O database to process an I/O request.

Format

status = driver_FDT_routine (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
pcb	PCB	input	reference
ucb	UCB	input	reference
ccb	CCB	input	reference

irp
I/O request packet for the current I/O request. An FDT routine may read the following IRP fields:

Field	Contents
IRP$L_FUNC	I/O function code supplied in the $QIO request
IRP$L_QIO_P*n*	Function-specific $QIO system service arguments (**p1** through **p6**); *n* corresponds to an integer from 1 to 6.

pcb
Process control block of the current process

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request

ccb
Channel control block that describes the process-I/O channel

Essentials

Identifying the Routine
Use the FDT_ACT macro to insert the procedure value of an upper-level FDT action routine into the FDT action routine vector slot that corresponds to a specified I/O function code.

Declaring the Entry Point
Use:

$DRIVER_FDT_ENTRY [preserve=<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15>]
[,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the upper-level FDT action routine.
fetch=YES, the default, loads the addresses of the IRP, PCB, UCB, and CCB into R3, R4, R5, and R6, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver upper-level FDT action routine that uses this macro can access any of its arguments by using a symbolic name of the form FDTARG$**_argument-name**.

♦

Called by

The $QIO system service calls a driver's upper-level FDT action routine from the module SYSQIOREQ. An upper-level FDT action routine can call any number of FDT support routines, as long as each returns control and status to the upper-level routine.

Context

An FDT routine is called at IPL$_ASTDEL and must exit at IPL$_ASTDEL. An FDT routine must not lower IPL below IPL$_ASTDEL. If it raises IPL, it must lower it to IPL$_ASTDEL before passing control to any other code. Similarly, before exiting, it must release any spin locks it may have acquired in an OpenVMS multiprocessing environment.

FDT routines execute in the context of the process that requested the I/O activity. If an FDT routine alters the stack, it must restore the stack before returning control to the caller of the routine.

Exit mechanism

An FDT routine must return control and status to its caller. An upper-level FDT action routine returns SS$_FDT_COMPL status to the $QIO system service and passes the return status to be delivered to the caller of $QIO in the FDT_CONTEXT structure.

Description

An upper-level FDT routine (and any FDT support routine it may call) validates the function-dependent arguments to a $QIO system service request and prepares the I/O database to service the request. For each function that a device supports, an upper-level FDT action routine must provide preprocessing of requests for that function. FDT processing may complete a a function that does not involve an I/O transfer. Otherwise FDT processing can abort the request or deliver it to the driver.

An OpenVMS AXP upper-level FDT action routine can invoke the $FDTARGDEF macro, defined in SYS$LIBRARY:LIB.MLB, to provide symbolic names for the standard AP offsets of the four parameters provided as input (IRP, PCB, UCB, and CCB) to all upper-level FDT action routines. A routine that does so can use names of the form FDTARG$_*xxx*, where *xxx* is the 3-letter structure acronym, to access the input parameters.

Step 1 OpenVMS AXP Device Driver Entry Points
FDT Upper-Level Action Routine

Notes for Converting Step 1 Drivers

- You must indicate the entry point of each upper-level FDT action routine with a $DRIVER_FDT_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

- You should examine an FDT routine's use of R0, R7, and R8.

  The FDT routine of an OpenVMS VAX device driver may obtain the address of FDT routine being called from R0, the number of the bit that specifies the code for the requested I/O function from R7, and the address of the entry in the function decision table that dispatched control to this FDT routine.

  An OpenVMS AXP driver can obtain the user-supplied function code from IRP$L_FUNC. It can obtain the address of the start of the FDT from DDT$PS_FDT2. The DDT address is available from UCB$L_DDT.

- An FDT routine of an OpenVMS VAX device driver accesses values of the function-dependent arguments specified in the $QIO request as offsets from the value of the AP; an OpenVMS AXP device driver obtains them from the IRP (at symbolic offsets IRP$L_QIO_P1 through IRP$L_QIO_P6).

FDT Error-Handling Callback Routine

Processes error conditions that occur during EXE_STD$READLOCK, EXE_STD$WRITELOCK, and EXE_STD$MODIFYLOCK processing.

Format

status = error_callback (irp, pcb, ucb, ccb, status)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
pcb	PCB	input	reference
ucb	UCB	input	reference
ccb	CCB	input	reference
status	integer	input	value

irp

I/O request packet for the current I/O request.

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb

Channel control block that describes the process-I/O channel.

status

Error status returned by buffer accessibility check (SS$_ACCVIO or SS$_BADPARAM) or buffer locking operation (SS$_ACCVIO, SS$_INSFWSL, or page fault status).

Essentials

Identifying the Routine

Use the **errtn** argument in a call to EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, or EXE_STD$WRITELOCK.

Declaring the Entry Point

Use:

$DRIVER_ERRRTN_ENTRY [preserve=<>] [,fetch=YES] where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the upper-level FDT action routine
fetch=YES, the default, loads the addresses of the IRP, PCB, UCB, CCB, and status into R3, R4, R5, R6, and R0, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver error-handling callback routine that uses this macro can access any of its arguments by using a symbolic name of the form ERRARG$**_argument-name**.

♦

Called by

EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, and EXE_STD$WRITELOCK call the driver's error-handling callback routine to process errors incurred by a buffer accessibility check or buffer locking operation.

Context

An error-handling callback routine is called at IPL$_ASTDEL and must exit at IPL$_ASTDEL. An error-handling callback routine must not lower IPL below IPL$_ASTDEL. If it raises IPL, it must lower it to IPL$_ASTDEL before passing control to any other code. Similarly, before exiting, it must release any spin locks it may have acquired in an OpenVMS multiprocessing environment.

Error-handling callback routines execute in the context of the process that requested the I/O activity. If a routine alters the stack, it must restore the stack before returning control to the caller of the routine.

Exit mechanism

An error-handling callback routine must return control to its caller and preserve the contents of R0 and R1.

Description

An error-handling callback routine processes any errors incurred by a call to EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, or EXE_STD$WRITELOCK.

A driver typically requires an error-handling callback routine if it must lock multiple areas into memory for a single I/O request and must regain control, if the request is to be aborted, to unlock these areas. The routine performs such operations as locating the addresses of the previously allocated buffers (typically stored in the IRP) and calling MMG_STD$UNLOCK to release them.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of each FDT error-handling callback routine with a $DRIVER_ERRRTN_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

- You should examine an FDT routine's use of R0, R7, and R8.

The FDT routine of an OpenVMS VAX device driver may obtain the address of FDT routine being called from R0, the number of the bit that specifies the code for the requested I/O function from R7, and the address of the entry in the function decision table that dispatched control to this FDT routine.

An OpenVMS AXP driver can obtain the user-supplied function code from IRP$L_FUNC. It can obtain the address of the start of the FDT from DDT$PS_FDT2. The DDT address is available from UCB$L_DDT.

- An FDT routine of an OpenVMS VAX device driver accesses values of the function-dependent arguments specified in the $QIO request as offsets from the value of the AP; an OpenVMS AXP device driver obtains them from the IRP (at symbolic offsets IRP$L_QIO_P1 through IRP$L_QIO_P6).

Interrupt Service Routine

Processes interrupts generated by a device.

Format

DRIVER_INTERRUPT (idb)

Arguments

Argument	Type	Access	Mechanism
idb	IDB	input	reference

idb
Interrupt dispatch block.

Essentials

Identifying the Routine
Devices require an interrupt service routine for each interrupt vector. Use the DPT_STORE_ISR macro to store the ISR procedure descriptor and entry point address in the interrupt transfer vector block (VEC) at CRB$L_INTD. You can find the second and third VECs at CRB$L_INTD2 and CRB$L_INTD+2*VEC$K_LENGTH, respectively.

Declaring the Entry Point
Indicate the entry point of an OpenVMS AXP interrupt service routine with a .CALL_ENTRY Macro-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored. If the interrupt service routine forks, transferring control to a fork routine, it must declare, at its .CALL_ENTRY point, R3, R4, and R5 as **input** registers.

Called by
The interrupt service routine is called either by the OpenVMS interrupt dispatcher (for direct-vectored adapters) or by an adapter interrupt service routine (for non-direct-vector adapters).

Context
An OpenVMS AXP driver's interrupt service routine conforms to the OpenVMS calling standard. As such, it need not explicitly save any registers that are saved on its behalf by the compiler. It must, however, preserve any of the non-scratch registers R8 through R15 if it uses them.

An interrupt service routine is called, executes, and returns at device IPL. It must obtain the device lock associated with its device IPL. It performs this acquisition as soon as it obtains the address of the UCB of the interrupting device. It must release this device lock before dismissing the interrupt.

At the execution of a driver's interrupt service routine, the processor is running in interrupt mode on the kernel stack. As a result, an interrupt service routine can reference only those virtual addresses that reside in system (S0) space.

Exit mechanism

Before an interrupt service routine transfers control to the suspended driver, it must restore (using EVAX_LDQ MACRO-32 compiler built-ins) the contents of R3 and R4 from UCB$Q_FR3 and UCB$Q_FR4. It then issues a JSB instruction to the procedure value saved in UCB$L_FPC.

When it regains control (after the suspended driver forks), the interrupt service routine returns to the interrupt dispatcher.

Description

An interrupt service routine performs the following functions:

1. Determines whether the interrupt is expected.
2. Processes or dismisses unexpected interrupts.
3. Activates the suspended driver so it can process expected interrupts.

Notes for Converting Step 1 Drivers

- An OpenVMS VAX device driver declares an interrupt service routine by issuing the DPT_STORE macro to store its address in an interrupt transfer vector block. Because the OpenVMS AXP interrupt dispatcher requires the addresses of both the code entry point and the procedure descriptor of an interrupt service routine, you must use the new DPT_STORE_ISR macro (which generates both) to declare the routine.
- The OpenVMS VAX interrupt dispatcher issues a JSB instruction to pass control to an OpenVMS VAX driver's interrupt service routine; the OpenVMS AXP interrupt dispatcher issues a standard call to a driver's interrupt service routine. This results in some substantial differences:
  - You must indicate the entry point of an OpenVMS AXP interrupt service routine with a .CALL_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored.
  - An OpenVMS VAX driver's interrupt service routine must preserve R6 through R11, the AP, and the FP, if it uses them.

    An OpenVMS AXP device driver's interrupt service routine declares at its entry point those registers it passes as input to its fork routine. For instance:

    ```
    .CALL_ENTRY input=<R3,R4,R5>).
    ```

    As a routine that conforms to the OpenVMS calling standard, it need not explicitly save any registers that are saved on its behalf by the compiler. It must, however, preserve any of the non-scratch registers R8 through R15 if it uses them.
  - An OpenVMS VAX driver's interrupt service routine is passed various information on the stack, including the address of the IDB, the contents of R0 through R5, the PC, and PSL at the time of the interrupt.

    The only parameter passed to an OpenVMS AXP driver's interrupt service routine is the address of the IDB (that is, the contents of VEC$L_IDB). The routine cannot reference data on the stack.

- Before exiting, an OpenVMS VAX driver's interrupt service routine removes the address of the pointer to the IDB from the top of the stack and restores the registers OpenVMS saved when dispatching the interrupt.

  An OpenVMS AXP driver's interrupt service routine does not perform these actions.

- An OpenVMS VAX driver's interrupt service routine exits with an REI instruction.

  An OpenVMS AXP driver's interrupt service routine exits by returning control to its caller.

- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- If you alter the driver's suspension mechanism such that it uses the OpenVMS kernel process services, you must change the mechanism by which the interrupt service routine reactivates lower IPL execution threads by replacing the IOFORK macro with the KP_STALL_IOFORK macro.

Mount Verification Routine

Performs device-specific mount verification.

Format

MNTVER (irp, ucb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
ucb	UCB	input	reference

irp
I/O request packet, or zero to complete mount verification.

ucb
Unit control block.

Essentials

Identifying the Routine

Supply the address of the mount verification routine in the **mntver** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT. The default value of this argument, IOC_STD$MNTVER, is the only value allowed non-Digital-supplied device drivers.

Declaring the Entry Point

Use:

$DRIVER_MNTVER_ENTRY [preserve=<>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel selective routine
fetch=YES, the default, loads the addresses of IRP and UCB into R3 and R5, respectively, **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver mount verification routine that uses this macro can access any of its arguments by using a symbolic name of the form MNTARG$**_argument-name**.

♦

Called by

Routine DRIVER_CODE in module MOUNTVER calls a driver's mount verification routine.

Context

A mount verification routine is called at fork IPL with the corresponding fork lock held in a multiprocessing system.

Exit mechanism

The mount verification routine returns to its caller.

Description

Reserved to Digital.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of a mount verification routine with a $DRIVER_MNTVER_ENTRY indicating which registers must be saved and restored across routine execution.

♦

Register Dumping Routine

Copies the contents of a device's registers to an error message buffer or a diagnostic buffer.

Format

status = REGDMP (buffer, arg_2, ucb)

Arguments

Argument	Type	Access	Mechanism
buffer	address	input	reference
arg_2	unspecified	input	reference
ucb	UCB	input	reference

buffer
Address of buffer into which a register dumping routine copies the contents of device registers).

arg_2
Device-specific argument, usually a controller register access mailbox (CRAM).

ucb
Unit control block.

Essentials

Identifying the Routine
Specify the name of the register dumping routine in the **regdmp** argument of the DDTAB macro. This macro places the procedure value of the routine into the DDT.

Declaring the Entry Point

Use:

$DRIVER_REGDUMP_ENTRY [preserve=<R2>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the register dumping routine
fetch=YES, the default, loads the addresses of the buffer, the driver-specific argument. and the UCB into R0, R4, and R5, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver register dumping routine that uses this macro can access any of its arguments by using a symbolic name of the form REGARG$**_argument-name**.

♦

Step 1 OpenVMS AXP Device Driver Entry Points
Register Dumping Routine

Called by
The system error logging routines (ERL_STD$DEVICERR, ERL_STD$DEVICTMO, and ERL_STD$DEVICEATTN) and diagnostic buffer filling routine (IOC_STD$DIAGBUFILL) call the register dumping routine.

Context
OpenVMS calls a register dumping routine at the same IPL at which the driver called the OpenVMS AXP system routine ERL_STD$DEVICERR, ERL_STD$DEVICTMO, ERL_STD$DEVICEATTN, or IOC_STD$DIAGBUFILL. A register dumping routine must not change IPL.

A register dumping routine executes within the context of an interrupt service routine or a driver fork process, using the kernel-mode stack. As a result, it can only refer to those virtual addresses that reside in system (S0) space. If it uses the stack, the register dumping routine must restore the stack before passing control to another routine, waiting for an interrupt, or returning control to its caller.

Exit mechanism
The register dumping routine returns to its caller.

Description

A register dumping routine fills the indicated buffer as follows:

1. Writes a longword value representing the number of device registers to be written into the buffer
2. Moves device register longword values into the buffer following the register count longword

Notes for Converting Step 1 Drivers

- You must indicate the entry point of the routine with a $DRIVER_REGDUMP_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

- An OpenVMS VAX device driver's register-dumping routine is passed the address of the device's CSR in R4 (if the driver invoked the WFIKPCH macro to wait for an interrupt or timeout).

  An OpenVMS AXP device driver's register-dumping routine is not passed the address of the CSR. It may access the controller's register by means of the controller register access mailbox (CRAM), the address of which is usually passed in **arg_2**.

- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.

Start-I/O Routine (Traditional)

Activates a device to process a requested I/O function.

Format

START (irp, ucb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
ucb	UCB	input	reference

irp
I/O request packet.

ucb
Unit control block. The start-I/O routine uses information from the following UCB fields to calculate the size and location of a transfer:

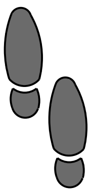
Field	Description
UCB$L_BCNT	Number of bytes to be transferred, copied from the low-order word of IRP$L_BCNT
UCB$L_BOFF	Byte offset into first page of direct-I/O transfer; for buffered-I/O transfers, number of bytes to be charged to the process allocating the buffer
UCB$L_SVAPTE	For a *direct-I/O* transfer, virtual address of first page-table entry (PTE) of I/O-transfer buffer; for *buffered-I/O* transfer, address of buffer in system address space

Essentials

Identifying the Routine

Specify the name of the start-I/O routine in the **start** argument of the DDTAB macro. This macro places the address of the routine into the DDT.

Declaring the Entry Point



Use:

$DRIVER_START_ENTRY [preserve=<R2,R4>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the start-I/O routine

fetch=YES, the default, loads the addresses of the IRP and UCB into R3 and R5, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver start-I/O routine that uses this macro can access any of its arguments by using a symbolic name of the form STARTARG$_**argument-name**.

♦

Called by

A traditional start-I/O routine is called as the result of a standard call issued by IOC_STD$INITIATE and IOC_STD$REQCOM in module IOSUBNPAG.

Context

A start-I/O routine is placed into execution at fork IPL, holding the associated fork lock. It must relinquish control of the processor in the same context.

For many devices, the start-I/O routine raises IPL to IPL$_POWER to check that a power failure has not occurred on the device prior to loading the device's registers. The start-I/O routine initiates device activity at device IPL, after acquiring the corresponding device lock. An invocation of the WFIKPCH or WFIRLCH macro (or KP_STALL_WFIKPCH or KP_STALL_WFIRLCH) to wait for a device interrupt releases this device lock.

Because a start-I/O routine gains control of the processor in the context of a fork process, it can refer only to those addresses that reside in system (S0) space. If the start-I/O routine uses the stack, it must restore the stack before completing the request, waiting for an interrupt, or requesting system resources.

Exit mechanism

A traditional start-I/O routine suspends itself whenever it must wait for a required resource, such as a controller data channel. To do so, it invokes a OpenVMS macro (such as REQPCHAN) that saves its context in the UCB fork block, places the UCB in a resource wait queue, and returns control to the caller of the start-I/O routine.

The start-I/O routine also suspends itself when it issues a WFIKPCH or WFIRLCH macro to initiate device activity. These macros also store the driver's context in the UCB fork block to be restored when the device interrupts or times out.

The start-I/O routine is again suspended if it forks to complete servicing of a device interrupt. The IOFORK macro places driver context in the UCB fork block, inserts the fork block into a processor-specific fork queue, and requests a software interrupt from the processor at the corresponding fork IPL. After issuing an IOFORK macro, the routine returns control to the driver's interrupt service routine.

The routine completes the processing of an I/O request by invoking the REQCOM macro. In addition to initiating device-independent postprocessing of the current request, the REQCOM macro also attempts to start the next request waiting for a device unit. If there are no waiting requests, the macro returns control to the caller of the start-I/O routine. For traditional start-I/O routines, this is often the OpenVMS fork dispatcher.

Description

A driver's start-I/O routine activates a device and waits for a device interrupt or timeout. After a device interrupt, the driver's interrupt service routine returns control to the start-I/O routine at device IPL, holding the associated device lock.

The start-I/O routine usually forks at this time to perform various device-dependent postprocessing tasks, and returns control to the interrupt service routine.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of the start-I/O routine with a $DRIVER_START_ENTRY macro, indicating which registers must be saved and restored across routine execution.

  ♦

- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- You should examine the routine's use of suspension mechanisms (for instance, its forking, wait-for-interrupt, and resource-wait semantics) to determine whether it needs to be adapted to use the kernel process services. Typically a driver that makes subroutine calls before suspending itself (and relies on the previous context of these subroutines remaining intact on the stack), must be adapted to use the kernel process services.

Start-I/O Routine (Kernel Process)

Activates a device to process a requested I/O function.

Format

START (kpb)

Arguments

Argument	Type	Access	Mechanism
kpb	KPB	input	reference

kpb
Kernel process block.

Essentials

Identifying the Routine
Specify the name of the kernel process start-I/O routine (EXE$KP_STARTIO) in the **start** argument of the DDTAB macro, and the name of the driver's start-I/O routine in the **kp_startio** argument.

Declaring the Entry Point
Indicate the entry point of a kernel process start-I/O routine with a .CALL_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored.

Called by
A kernel-process start-I/O routine is called by EXE$KP_STARTIO in module KERNEL_PROCESS.

Context
A kernel process start-I/O routine is placed into execution at fork IPL, holding the associated fork lock. It must relinquish control of the processor in the same context.

For many devices, the start-I/O routine raises IPL to IPL$_POWER to check that a power failure has not occurred on the device prior to loading the device's registers. The start-I/O routine initiates device activity at device IPL, after acquiring the corresponding device lock. An invocation of the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro to wait for a device interrupt releases this device lock.

Because a start-I/O routine gains control of the processor in the context of a fork process, it can refer only to those addresses that reside in system (S0) space.

Neither the start-I/O routine that initiates a kernel process nor the kernel process thread can depend on inheriting the synchronization capabilities (such as spin locks and IPL) of the other when control is exchanged betwen them. If they must share data or perform other operations that require synchonization, they must explicitly establish a synchronization mechanism.

The kernel process cannot assume that its initiator is not running in parallel, nor can the initiator of the kernel process assume that the kernel process has already executed when EXE_STD$KP_START returns control.

Exit mechanism
A kernel process start-I/O routine suspends itself whenever it must wait for a required resource, such as a controller data channel. To do so, it invokes an OpenVMS macro (such as KP_STALL_REQCHAN) that saves its context in the UCB fork block, places the UCB in a resource wait queue, and returns control to the caller of the start-I/O routine.

The start-I/O routine also suspends itself when it issues a KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro to initiate device activity. These macros also store the driver's context in the UCB fork block to be restored when the device interrupts or times out.

The start-I/O routine is again suspended if it forks to complete servicing of a device interrupt. The KP_STALL_IOFORK macro places driver context in the UCB fork block, inserts the fork block into a processor-specific fork queue, and requests a software interrupt from the processor at the corresponding fork IPL. After issuing a KP_STALL_IOFORK macro, the routine issues an RSB instruction, returning control to the driver's interrupt service routine.

The routine completes the processing of an I/O request by invoking the KP_ REQCOM macro. In addition to initiating device-independent postprocessing of the current request, the KP_REQCOM macro also attempts to start the next request waiting for a device unit. If there are no waiting requests, the macro returns control to the caller of the kernel process start-I/O routine, EXE$KP_ STARTIO.

Description

A driver's start-I/O routine activates a device and waits for a device interrupt or timeout. After a device interrupt, the driver's interrupt service routine returns control to the start-I/O routine at device IPL, holding the associated device lock.

The start-I/O routine usually forks at this time to perform various device-dependent postprocessing tasks, and returns control to the interrupt service routine.

Notes for Converting Step 1 Drivers

- If the routine need not be converted to a kernel process, you must replace any calls to EXE$FORK, EXE$FORK_WAIT, EXE$IOFORK, IOC$WFIKPCH, IOC$WFIRLCH, IOC$REQPCHANH, and IOC$REQPCHANL with invocations of the appropriate suspension macro or with calls to EXE_ STD$PRIMITIVE_FORK, IOC_STD$PRIMITIVE_WFIKPCH, IOC_ STD$PRIMITIVE_WFIRLCH, IOC_STD$PRIMITIVE_REQCHANH, or IOC_STD$PRIMITIVE_REQCHANL.
- You must indicate the entry point of a kernel process start-I/O routine with a .CALL_ENTRY Macro-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored. A kernel process start-I/O routine invokes the KP_REQCOM macro (in place of the REQCOM macro) to return control properly to its caller.

Timeout Handling Code (Traditional)

Takes whatever action is necessary when a device has not yet responded to a request for device activity, and the time allowed for a response has expired.

Format

BNEQ timeout-code-address

Input

Location	**Contents**
R3	Contents of R3 when the last invocation of WFIKPCH or WFIRLCH took place (usually the address of the IRP)
R4	Contents of R4 when the last invocation of WFIKPCH or WFIRLCH took place (usually the address of the IDB)
R5	Address of UCB of the device
UCB$L_STS	UCB$V_INT and UCB$V_TIM clear; UCB$V_TIMOUT set

Essentials

Identifying the Timeout Handler

Specify the address of timeout code in the **excpt** argument to the WFIKPCH or WFIRLCH macro.

Branched to

The WFIKPCH and WFIRLCH macros use this entry point, but only when the label of timeout code is provided in their **excpt** argument. These macros are used in the driver's start-I/O routine; thus, strictly speaking, the driver itself is the only entity that uses this entry point.

The OpenVMS AXP software timer interrupt service routine restarts a stalled driver fork procedure, passing to it a status (UCB$V_TIMOUT in UCB$L_STS) which is inspected by one of two instructions left at the top of the fork procedure by the WFIKPCH or WFIRLCH macro. If UCB$V_TIMOUT is set, the second instruction branches to the timeout code.

Context

Timeout code receives control at device IPL and must exit at device IPL. At the time the timeout code executes, the processor holds both the fork lock and the device lock associated with the device.

After taking whatever device-specific action is necessary at device IPL, timeout code can lower IPL to fork IPL to perform less critical activities. Because its caller restores IPL to fork IPL (and releases the device lock), if a timeout handler lowers IPL, it can do so only by forking or by performing the following steps:

1. Issue a DEVICEUNLOCK macro to lower to fork level
2. Perform timeout handling activities possible at the lower IPL

3. Issue a DEVICELOCK macro to again obtain the device lock and raise to device IPL

Timeout code can access only those virtual addresses that refer to system (S0) space.

Traditional timeout code can use R0, R1, and R2 freely, but must preserve the contents of all other registers. If it uses the stack, it must restore the stack before completing or canceling the current I/O request, waiting for an interrupt, or returning control to its caller.

Exit mechanism

Traditional timeout code issues an RSB instruction to return to the software timer interrupt service routine, restarts the I/O request, or invokes the REQCOM macro to complete the I/O request that encountered the timeout.

Description

There are no outputs required from timeout code but, depending on the characteristics of the device, timeout code might cancel or retry the current I/O request, send a message to the operator, or take some other action.

Before timeout code executes, the system has placed the device in a state in which no interrupt is expected (by clearing the bit UCB$V_INT in field UCB$L_STS). If the requested interrupt occurs while this routine executes, it will appear to be an unsolicited interrupt. Many drivers handle this situation by disabling interrupts while timeout code executes.

Notes for Converting Step 1 Drivers

- On OpenVMS VAX systems, the software timer interrupt service routine issues a JSB instruction to a timeout handling routine within a driver when it detects that a device has timed out. On OpenVMS AXP systems, the OpenVMS AXP suspension macros provide a mechanism by which the driver fork routine, when resumed by a timeout, tests the timeout bit in the UCB and branches, if the bit is set, to the address of the timeout code.
- You must replace direct control and status register (CSR) access (for instance, by means of a MOVL instruction) with CSR access by means of a controller register access mailbox (CRAM).

Timeout Handling Code (Kernel Process)

Takes whatever action is necessary when a device has not yet responded to a request for device activity, and the time allowed for a response has expired.

Format

BLBC timeout-code-address

Arguments

None.

Essentials

Identifying the Routine

Specify the address of the timeout code in the **excpt** argument to the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro.

Branched to

The KP_STALL_WFIKPCH, and KP_STALL_WFIRLCH macros use this entry point, but only when the label of timeout code is provided in their **excpt** argument. These macros are used in the driver's start-I/O routine; thus, strictly speaking, the driver itself is the only entity that uses this entry point.

The OpenVMS AXP software timer interrupt service routine restarts a stalled driver kernel process fork procedure, passing a status (UCB$V_TIMOUT in UCB$L_STS) to it, which is inspected by one of two instructions left at the top of the fork procedure by the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro. If UCB$V_TIMOUT is set, the second instruction branches to the timeout code.

Context

The timeout code receives control at device IPL and must exit at device IPL. At the time the timeout code executes, the processor holds both the fork lock and device lock associated with the device.

After taking whatever device-specific action is necessary at device IPL, timeout code can lower IPL to fork IPL to perform less critical activities. Because its caller restores IPL to fork IPL (and releases the device lock), if timeout code lowers IPL, it can do so only by forking or by performing the following steps:

1. Issue a DEVICEUNLOCK macro to lower to fork level
2. Perform timeout handling activities possible at the lower IPL
3. Issue a DEVICELOCK macro to again obtain the device lock and raise to device IPL

Timeout code can access only those virtual addresses that refer to system (S0) space.

Kernel process timeout code executes in the context of the kernel process thread that invoked the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro.

Exit mechanism

Kernel process timeout code executes as part of the kernel process thread that invoked WFIKPCH or WFIRLCH macro and therefore has no special exit mechanism.

Description

There are no outputs required from timeout code but, depending on the characteristics of the device, timeout code might cancel or retry the current I/O request, send a message to the operator, or take some other action.

Before timeout code executes, OpenVMS has placed the device in a state in which no interrupt is expected (by clearing the bit UCB$V_INT in field UCB$L_STS). If the requested interrupt occurs while this routine executes, it will appear to be an unsolicited interrupt. Many drivers handle this situation by disabling interrupts while timeout code executes.

Notes for Converting Step 1 Drivers

- On OpenVMS VAX systems, the software timer interrupt service routine issues a JSB instruction to a timeout handling routine within a driver when it detects that a device has timed out. On OpenVMS AXP systems, the OpenVMS AXP suspension macros provide a mechanism by which the driver fork routine, when resumed by a timeout, tests the timeout bit in the UCB and branches, if the bit is set, to the address of the timeout code.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.

Unit Delivery Routine

For controllers that can control a variable number of device units, determines which specific devices are present and available for inclusion in the system's configuration.

Format

status = DELIVER (ddb, idb, unit_number, scratch_area, adp)

Arguments

Argument	Type	Access	Mechanism
ddb	DDB	input	reference
idb	IDB	input	reference
unit_number	integer	input	value
scratch_area	address	input	reference
adp	ADP	input	reference

ddb
Device data block.

idb
Interrupt dispatch block; 0 if none exists.

unit_number
Number of unit that the unit delivery routine must decide to configure or not to configure.

scratch_area
Address of quadword scratch area.

adp
Adapter control block.

Essentials

Identifying the Routine
Specify the name of the unit delivery routine in the **deliver** argument to the DPTAB macro. The macro puts the procedure value address of this routine in the DPT.

Declaring the Entry Point

Use:

$DRIVER_DELIVER_ENTRY [preserve=<R2>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the unit delivery routine

fetch=YES, the default, loads the address of the IDB into R3 and R4, the unit number into R5, the address of the scratch area into R7, and the address of the ADP into R8; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver unit delivery routine that uses this macro can access any of its arguments by using a symbolic name of the form DLVRARG$**_argument-name**.

♦

Called by

The System Management (SYSMAN) Utility's IO AUTOCONFIGURE command calls the unit delivery routine once for each unit the controller is capable of controlling. This value is specified in the **defunits** argument to the DPTAB macro.

Context

The unit delivery routine is called at IPL$_POWER. It must not lower IPL. The unit delivery routine executes in the context of the process within which the autoconfiguration facility executes.

Exit mechanism

A unit delivery routine returns success or failure status to the autoconfiguration facility. If the routine returns error status, the unit is not configured.

Description

The unit delivery routine determines which units on a controller should be configured. For instance, a unit delivery routine can prevent the creation of UCBs for devices that do not respond to a test for their presence.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of the routine with a $DRIVER_DELIVER_ENTRY macro to indicate which registers must be saved and restored across routine execution.

  ♦

- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- The unit delivery routine of an OpenVMS VAX device driver receives the addresses of the device CSR in R4 and the IDB in R5. An OpenVMS AXP device driver's unit delivery routine is not passed the address of the CSR. It may access the controller's register by means of the controller register access mailbox (CRAM), the address of which is provided in IDB$PS_CRAM.

Step 1 OpenVMS AXP Device Driver Entry Points
Unit Delivery Routine

- An OpenVMS AXP unit delivery routine is passed the address of the device data block and the address of a quadword scratch area.

♦

Unit Initialization Routine

Prepares a device for operation and, in the case of a device on a dedicated controller, initializes the controller.

Format

status = UNITINIT (idb, ucb)

Arguments

Argument	Type	Access	Mechanism
idb	IDB	input	reference
ucb	UCB	input	reference

idb
Interrupt dispatch block associated with the controller.

ucb
Unit control block.

Essentials

Identifying the Routine

Specify the address of the unit initialization routine **unitinit** argument of the DDTAB macro. This macro places the procedure value of the routine into the DDT.

Declaring the Entry Point

Use:

$DRIVER_UNITINIT_ENTRY [preserve=<R2>] [,fetch=YES]

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the unit initialization routine
fetch=YES, the default, loads $SS_NORMAL status into R0, and the addresses of the IDB and UCB into R4 and R5, respectively; **fetch=NO** disables register loading. Regardless of the value of the **fetch** argument, a driver unit initialization routine that uses this macro can access any of its arguments by using a symbolic name of the form UNITARG$_**argument-name**.

♦

Called by

The driver-loading procedure calls a driver's unit initialization routine when processing a CONNECT command. OpenVMS calls a unit initialization routine when the device, the controller, the processor, or the adapter to which the device is connected undergoes power failure recovery.

Context

OpenVMS calls a unit initialization routine at IPL$_POWER. If it must lower IPL, the controller initialization routine cannot explicitly do so. Rather, it must fork. Because the driver-loading procedure calls the unit initialization routine immediately after the controller initialization returns control to it, the driver's initialization routines must synchronize their activities.

The portion of the unit initialization routine that services power failure cannot acquire any spin locks. As a result, the routine cannot fork to perform power failure servicing.

Because OpenVMS calls it in system context, a unit initialization routine can only refer to those virtual addresses that reside in system (S0) space. R0, and preserve the contents of all registers except R0, R1, and R2.

Exit mechanism

A unit initialization routine returns success or failure status to its caller.

Description

Depending on the device, a unit initialization routine performs any or all of the following tasks:

1. Determines whether it is being called as a result of a power failure by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB. A unit initialization routine may want to perform or avoid specific tasks when servicing a power failure.
2. Clears error-status bits in device registers.
3. Enables controller interrupts.
4. Sets the online bit (UCB$V_ONLINE in UCB$L_STS).
5. Allocates resources that must be permanently allocated to the device or, for some devices, the controller.
6. If the device has a dedicated controller, as some printers do, fills in IDB$PS_OWNER.
7. For dedicated controllers, initializes controller and device hardware.

Notes for Converting Step 1 Drivers

- You must indicate the entry point of the routine with a .JSB_ENTRY Macro-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored.

  ♦

- An OpenVMS VAX device driver can specify a controller initialization routine by invoking the DPT_STORE macro to place its address into the interrupt transfer vector block (CRB$L_INTD+VEC$L_UNITINIT). An OpenVMS AXP device driver specifies the routine in the **unitinit** argument of the DDTAB macro.

- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- The unit initialization routine of an OpenVMS VAX device driver receives the addresses of the primary and secondary device CSRs in R3 and R4, respectively. An OpenVMS AXP device driver's unit initialization routine is not passed the addresses of the CSRs. It may access the controller registers by means of the controller register access mailbox (CRAM), the address of which is provided in IDB$PS_CRAM.
- An OpenVMS AXP unit initialization routine must return success or failure status to its caller.

Index

A

ACB (AST control block), 2–16 to 2–17, 2–18 to 2–19, 2–26 to 2–27
  contents, 2–32 to 2–34
ACB$V_QUOTA, 2–33
ACP_STD$ACCESS routine, 2–2 to 2–3
ACP_STD$ACCESSNET routine, 2–4 to 2–5
ACP_STD$DEACCESS routine, 2–6 to 2–7
ACP_STD$MODIFY routine, 2–8 to 2–9
ACP_STD$MOUNT routine, 2–10 to 2–11
ACP_STD$READBLK routine, 2–12 to 2–13
ACP_STD$WRITEBLK routine, 2–14 to 2–15
Alternate start I/O routine, 2–49 to 2–50
Alternate start-I/O routines, 3–3 to 3–5
AST (asynchronous system trap), 2–32 to 2–34, 2–35 to 2–37
  delivering, 2–16 to 2–17, 2–18 to 2–19, 2–45
  for aborted I/O request, 2–45
  process-requested, 2–33
Attention AST
  delivering, 2–16 to 2–17, 2–18 to 2–19
  disabling, 2–32 to 2–34
  enabling, 2–32 to 2–34
  flushing, 2–26 to 2–27

B

Buffer
  allocating, 2–47 to 2–49
  deallocating, 2–24 to 2–25
  locking, 2–70 to 2–73, 2–74 to 2–79, 2–96 to 2–99, 2–104 to 2–109, 2–117 to 2–120, 2–125 to 2–130, 2–203 to 2–204
  moving data to from system to user, 1–44, 2–171 to 2–173
  moving data to from user to system, 1–43, 2–168 to 2–170
  testing accessibility of, 2–70 to 2–73, 2–74 to 2–79, 2–96 to 2–99, 2–100 to 2–103, 2–104 to 2–109, 2–117 to 2–120, 2–121 to 2–124, 2–125 to 2–130
  unlocking, 2–205 to 2–206
BYTCNT (byte count) quota
  debiting, 2–48
BYTLM (byte limit) quota
  debiting, 2–48

C

CALL_ABORTIO macro, 1–2
CALL_ALLOCBUF macro, 1–3
CALL_ALLOCEMB macro, 1–4
CALL_ALLOCIRP macro, 1–3
CALL_ALTQUEPKT macro, 1–5
CALL_ALTREQCOM macro, 1–6
CALL_BROADCAST macro, 1–7
CALL_CANCELIO macro, 1–8
CALL_CARRIAGE macro, 1–9
CALL_CHKCREACCES macro, 1–10
CALL_CHKDELACCES macro, 1–10
CALL_CHKEXEACCES macro, 1–10
CALL_CHKLOGACCES macro, 1–10
CALL_CHKPHYACCES macro, 1–10
CALL_CHKRDACCES macro, 1–10
CALL_CHKWRTACCES macro, 1–10
CALL_CLONE_UCB macro, 1–11
CALL_COPY_UCB macro, 1–12
CALL_CREDIT_UCB macro, 1–13
CALL_CVTLOGPHY macro, 1–14
CALL_CVT_DEVNAM macro, 1–15
CALL_DELATTNAST macro, 1–16
CALL_DELATTNASTP macro, 1–17
CALL_DELCTRLAST macro, 1–18
CALL_DELCTRLASTP macro, 1–19
CALL_DELETE_UCB macro, 1–20
CALL_DEVICEATTN macro, 1–21
CALL_DEVICERR macro, 1–21
CALL_DEVICTMO macro, 1–21
CALL_DIAGBUFILL macro, 1–22
CALL_DRVDEALMEM macro, 1–23
CALL_FILSPT macro, 1–24
CALL_FINISHIO macro, 1–25
CALL_FINISHIOC macro, 1–25
CALL_FINISHIO_NOIOST macro, 1–25
CALL_FLUSHATTNS macro, 1–26
CALL_FLUSHCTRLS macro, 1–27
CALL_GETBYTE macro, 1–28
CALL_INITBUFWIND macro, 1–29
CALL_INITIATE macro, 1–30

CALL_INSERT_IRP macro, 1–31
CALL_IOLOCK macro, 1–32
CALL_IOLOCKR macro, 1–33
CALL_IOLOCKW macro, 1–34
CALL_IORSNWAIT macro, 1–35
CALL_IOUNLOCK macro, 1–36
CALL_LINK_UCB macro, 1–37
CALL_MAPVBLK macro, 1–38
CALL_MNTVER macro, 1–39
CALL_MNTVERSIO macro, 1–40
CALL_MODIFYLOCK macro, 1–41
CALL_MODIFYLOCK_ERR macro, 1–41
CALL_MOUNT_VER macro, 1–42
CALL_MOVFRUSER macro, 1–43
CALL_MOVFRUSER2 macro, 1–43
CALL_MOVTOUSER macro, 1–44
CALL_MOVTOUSER2 macro, 1–44
CALL_PARSDEVNAM macro, 1–45
CALL_POST macro, 1–46
CALL_POST_IRP macro, 1–47
CALL_POST_NOCNT macro, 1–46
CALL_PTETOPFN macro, 1–48
CALL_QIOACPPKT macro, 1–49
CALL_QIODRVPKT macro, 1–50
CALL_QNXTSEG1 macro, 1–51
CALL_QXQPPKT macro, 1–52
CALL_READCHK macro, 1–53
CALL_READCHKR macro, 1–53
CALL_READLOCK macro, 1–54
CALL_READLOCK_ERR macro, 1–54
CALL_RELCHAN macro, 1–55
CALL_RELEASEMB macro, 1–56
CALL_REQCOM macro, 1–57
CALL_SEARCHDEV macro, 1–58
CALL_SEARCHINT macro, 1–59
CALL_SETATTNAST macro, 1–60
CALL_SETCTRLAST macro, 1–61
CALL_SEVER_UCB macro, 1–62
CALL_SIMREQCOM macro, 1–63
CALL_SNDEVMSG macro, 1–64
CALL_THREADCRB macro, 1–65
CALL_UNLOCK macro, 1–66
CALL_WRITECHK macro, 1–67
CALL_WRITECHKR macro, 1–67
CALL_WRITELOCK macro, 1–68
CALL_WRITELOCK_ERR macro, 1–68
CALL_WRTMAILBOX macro, 1–69
Cancel I/O routine
    flushing ASTs in, 2–26 to 2–27
Cancel selective routines, 3–9 to 3–10
Cancel-I/O routines, 3–6 to 3–8
Channel assign routines, 3–11 to 3–12
Channel index number, 2–140
Cloned UCB routines, 3–13 to 3–15
COM_STD$DELATTNAST routine, 2–16 to 2–17
COM_STD$DELATTNASTP routine, 2–18 to 2–19
COM_STD$DELCTRLAST routine, 2–20 to 2–21
COM_STD$DELCTRLASTP routine, 2–22 to 2–23
COM_STD$DRVDEALMEM routine, 2–24 to 2–25
COM_STD$FLUSHATTNS, 2–33
COM_STD$FLUSHATTNS routine, 2–26 to 2–27
COM_STD$FLUSHCTRLS routine, 2–28 to 2–29
COM_STD$POST routine, 2–30 to 2–31
COM_STD$POST_NOCNT routine, 2–30 to 2–31
COM_STD$SETATTNAST routine, 2–32 to 2–34
COM_STD$SETCTRLAST routine, 2–35 to 2–37
Control AST
    disabling, 2–35 to 2–37
    enabling, 2–35 to 2–37
Controller initialization routines, 3–16 to 3–18

D

Data transfer
    zero byte count, 2–72, 2–98, 2–119
DDTAB macro, 1–70 to 1–73
Device
    disk, 2–113, 2–190
    tape, 2–190
Device affinity, 2–160
Device characteristics
    retrieving, 2–110 to 2–111
    setting, 2–112 to 2–114
Device controller data channel
    releasing, 1–94, 2–187 to 2–188
Device controller data channel wait queue, 2–187
Device unit
    operations count, 2–190
Diagnostic buffer, 2–160
    filling, 2–151 to 2–152
Direct I/O
    checking accessibility of process buffer for, 2–100 to 2–103, 2–121 to 2–124
    locking a process buffer for, 2–70 to 2–73, 2–74 to 2–79, 2–96 to 2–99, 2–104 to 2–109, 2–117 to 2–120, 2–125 to 2–130
    unlocking process buffer, 2–205 to 2–206
Disk driver, 2–65 to 2–67
DMA transfer
    for read operation, 2–96 to 2–99, 2–104 to 2–109
    for read/write operation, 2–70 to 2–73
    for write operation, 2–74 to 2–79, 2–117 to 2–120, 2–125 to 2–130
DPT$V_SVP, 2–169, 2–172
DPTAB macro, 1–74 to 1–78
DPT_STORE macro, 1–79 to 1–81
DPT_STORE_ISR macro, 1–82
Driver entry points, 3–1 to 3–47
Driver macros, 1–1 to 2–1
Driver unloading routines, 3–19
$DRIVER_ALTSTART_ENTRY macro, 3–3

$DRIVER_CANCEL_ENTRY macro, 3–7
$DRIVER_CANCEL_SELECTIVE_ENTRY macro, 3–9
$DRIVER_CHANNEL_ASSIGN_ENTRY macro, 3–11
$DRIVER_CLONEDUCB_ENTRY macro, 3–14
$DRIVER_CTRLINIT_ENTRY macro, 3–16
$DRIVER_DELIVER_ENTRY macro, 3–43
$DRIVER_ERRRTN_ENTRY macro, 3–23
$DRIVER_FDT_ENTRY macro, 3–21
$DRIVER_MNTVER_ENTRY macro, 3–29
$DRIVER_REGDUMP_ENTRY macro, 3–31
$DRIVER_START_ENTRY macro, 3–33
$DRIVER_UNITINIT_ENTRY macro, 3–45

E

ERL_STD$ALLOCEMB routine, 2–38 to 2–39
ERL_STD$DEVICEATTN routine, 2–40 to 2–42
ERL_STD$DEVICERR routine, 2–40 to 2–42
ERL_STD$DEVICTMO routine, 2–40 to 2–42
ERL_STD$RELEASEMB routine, 2–43
Error logging, 2–40 to 2–42
Error message buffer
  releasing, 2–190
Event flag
  handling for aborted I/O request, 2–45
EXE$ILLIOFUNC, 2–57 to 2–58
EXE$WRTMAILBOX routine, 2–131 to 2–132
EXE_STD$ABORTIO, 2–113
EXE_STD$ABORTIO system routine, 2–44 to 2–46
EXE_STD$ALLOCBUF routine, 2–47 to 2–49
EXE_STD$ALLOCIRP routine, 2–47 to 2–49
EXE_STD$ALTQUEPKT routine, 2–49 to 2–50
EXE_STD$CARRIAGE routine, 2–51
EXE_STD$CHKCREACCES routine, 2–52 to 2–53
EXE_STD$CHKDELACCES routine, 2–52 to 2–53
EXE_STD$CHKEXEACCES routine, 2–52 to 2–53
EXE_STD$CHKLOGACCES routine, 2–52 to 2–53
EXE_STD$CHKPHYACCES routine, 2–52 to 2–53
EXE_STD$CHKRDACCES routine, 2–52 to 2–53
EXE_STD$CHKWRTACCES routine, 2–52 to 2–53
EXE_STD$FINISHIO, 2–66, 2–67
EXE_STD$FINISHIO routine, 2–54 to 2–56, 2–114
EXE_STD$INSERT_IRP routine, 2–59 to 2–60
EXE_STD$INSIOQ, 2–91
EXE_STD$INSIOQ routine, 2–61 to 2–62
EXE_STD$INSIOQC routine, 2–61 to 2–62
EXE_STD$IORSNWAIT routine, 2–63 to 2–64
EXE_STD$LCLDSKVALID routine, 2–65 to 2–67
EXE_STD$MNTVERSIO routine, 2–68 to 2–69
EXE_STD$MODIFY routine, 2–70 to 2–73
EXE_STD$MODIFYLOCK, 2–205
EXE_STD$MODIFYLOCK routine, 2–74 to 2–79
EXE_STD$MOUNT_VER routine, 2–80 to 2–81
EXE_STD$ONEPARM routine, 2–82 to 2–83
EXE_STD$PRIMITIVE_FORK routine, 2–84 to 2–85
EXE_STD$PRIMITIVE_FORK_WAIT routine, 2–86 to 2–87
EXE_STD$QIOACPPKT routine, 2–88 to 2–89
EXE_STD$QIODRVPKT, 2–67, 2–72, 2–83, 2–98, 2–134
EXE_STD$QIODRVPKT routine, 2–90 to 2–92, 2–114, 2–119
EXE_STD$QXQPPKT routine, 2–94 to 2–95
EXE_STD$READ routine, 2–96 to 2–99
EXE_STD$READCHK routine, 2–100 to 2–103
EXE_STD$READLOCK, 2–205
EXE_STD$READLOCK routine, 2–104 to 2–109
EXE_STD$SENSEMODE routine, 2–110 to 2–111
EXE_STD$SETCHAR routine, 2–112 to 2–114
EXE_STD$SETMODE routine, 2–112 to 2–114
EXE_STD$SNDEVMSG routine, 2–115 to 2–116
EXE_STD$WRITE routine, 2–117 to 2–120
EXE_STD$WRITECHK routine, 2–121 to 2–124
EXE_STD$WRITELOCK, 2–205
EXE_STD$WRITELOCK routine, 2–125 to 2–130
EXE_STD$WRTMAILBOX routine, 2–116
EXE_STD$ZEROPARM, 2–133 to 2–134

F

FDT error handling callback routines, 3–23 to 3–25
FDT routine
  adjusting process quotas in, 2–48
  completing an I/O operation in, 2–54 to 2–56
  for direct I/O, 2–70 to 2–73, 2–96 to 2–99, 2–117 to 2–120
  for disk I/O, 2–65 to 2–67
  setting attention ASTs in, 2–32 to 2–34
  setting control ASTs in, 2–35 to 2–37
  unlocking process buffers in, 2–205
FDT routines, 3–20 to 3–22
FDT_ACT macro, 1–83 to 1–84
FDT_BUF macro, 1–85
FDT_INI macro, 1–24, 1–86
FORK macro, 1–87 to 1–88
Fork process
  creation by IOC_STD$INITIATE, 2–159 to 2–161
FORK_routine, 1–89
FORK_ROUTINE macro, 1–89
FORK_WAIT macro, 1–90 to 1–91
Full duplex device driver
  I/O completion for, 2–30 to 2–31

I

I/O postprocessing
for aborted I/O request, 2–45
for full duplex device driver, 2–30 to 2–31
for I/O request involving no device activity, 2–54 to 2–56
I/O postprocessing queue, 2–30 to 2–31, 2–190
I/O request
aborting, 2–44 to 2–46
canceling, 2–139 to 2–140
completing, 2–189 to 2–191, 2–199 to 2–200
with no parameters, 2–133 to 2–134
with one parameter, 2–82 to 2–83
Interrupt service routines, 3–26 to 3–28
IO$_AVAILABLE function
servicing, 2–67
IO$_PACKACK function
servicing, 2–66
IO$_SENSECHAR function
servicing, 2–110 to 2–111
IO$_SENSEMODE function
servicing, 2–110 to 2–111
IO$_SETCHAR function
servicing, 2–112 to 2–114
IO$_SETMODE function
servicing, 2–112 to 2–114
IO$_UNLOAD function
servicing, 2–67
IOC$RELCHAN routine, 1–94
IOC$REQCOM routine, 1–96
IOC_STD$ALTREQCOM routine, 2–135 to 2–136
IOC_STD$BROADCAST routine, 2–137 to 2–138
IOC_STD$CANCELIO routine, 2–139 to 2–140
IOC_STD$CLONE_UCB routine, 2–141 to 2–142
IOC_STD$COPY_UCB routine, 2–143 to 2–144
IOC_STD$CREDIT_UCB routine, 2–145
IOC_STD$CVTLOGPHY routine, 2–148 to 2–149
IOC_STD$CVT_DEVNAM routine, 2–146 to 2–147
IOC_STD$DELETE_UCB routine, 2–150
IOC_STD$DIAGBUFILL routine, 2–151 to 2–152
IOC_STD$FILSPT routine, 2–153 to 2–154
IOC_STD$GETBYTE routine, 2–155 to 2–156
IOC_STD$INITBUFWIND routine, 2–157 to 2–158
IOC_STD$INITIATE routine, 2–159 to 2–161
IOC_STD$IOPOST
unlocking process buffers, 2–205
IOC_STD$LINK_UCB routine, 2–162 to 2–163
IOC_STD$MAPVBLK routine, 2–164 to 2–165
IOC_STD$MNTVER routine, 2–166 to 2–167
IOC_STD$MOVFRUSER routine, 2–168 to 2–170
IOC_STD$MOVFRUSER2 routine, 2–168 to 2–170
IOC_STD$MOVTOUSER routine, 2–171 to 2–173
IOC_STD$MOVTOUSER2 routine, 2–171 to 2–173
IOC_STD$PARSDEVNAM routine, 2–174 to 2–175
IOC_STD$POST_IRP routine, 2–176
IOC_STD$PRIMITIVE_REQCHANH routine, 2–181 to 2–183
IOC_STD$PRIMITIVE_REQCHANL routine, 2–181 to 2–183
IOC_STD$PRIMITIVE_WFIKPCH routine, 2–184 to 2–186
IOC_STD$PRIMITIVE_WFIRLCH routine, 2–184 to 2–186
IOC_STD$PTETOPFN routine, 2–177 to 2–178
IOC_STD$QNXTSEG1 routine, 2–179 to 2–180
IOC_STD$RELCHAN routine, 2–187 to 2–188
IOC_STD$REQCOM, 2–48
IOC_STD$REQCOM routine, 2–189 to 2–191
IOC_STD$SEARCHDEV routine, 2–192 to 2–193
IOC_STD$SEARCHINT routine, 2–194 to 2–195
IOC_STD$SENSEDISK routine, 2–196 to 2–197
IOC_STD$SEVER_UCB routine, 2–198
IOC_STD$SIMREQCOM routine, 2–199 to 2–200
IOC_STD$THREADCRB routine, 2–201 to 2–202
IOFORK macro, 1–92 to 1–93
IOSB (I/O status block), 2–190
IPL$_ASTDEL, 2–64, 2–88, 2–91, 2–94
IPL$_IOPOST, 2–31, 2–45, 2–55, 2–190
IPL$_MAILBOX, 2–116, 2–131
IRP (I/O request packet)
insertion in pending-I/O queue, 2–59 to 2–60, 2–61 to 2–62
unlocking buffers specified in, 2–205
IRP$B_CARCON, 2–71, 2–97, 2–118
IRP$L_BCNT, 2–159, 2–160
IRP$L_BOFF, 2–159, 2–160
IRP$L_CHAN, 2–140
IRP$L_DIAGBUF, 2–159, 2–160
IRP$L_MEDIA, 2–83, 2–114, 2–134
IRP$L_PID, 2–140
IRP$L_SVAPTE, 2–160
IRP$V_DIAGBUF, 2–159, 2–160
IRP$V_FUNC, 2–97
IRPE (I/O request packet extension)
deallocation, 2–205
unlocking buffers specified in, 2–205

J

JIB$L_BYTCNT, 2–48
JIB$L_BYTLM, 2–48
Job controller
sending a message to, 2–116, 2–131 to 2–132
Job quota
byte count, 2–48
byte limit, 2–48

L

Local disk
  online count, 2–65
  valid bit, 2–65
Local disk UCB extension
  required for EXE_STD$LCLDSKVALID routine, 2–67
Logical I/O function
  translation to physical function, 2–70, 2–96, 2–117

M

Mailbox
  sending a message to, 2–115 to 2–116, 2–131 to 2–132
MAILBOX spin lock, 2–116, 2–131
MMG spin lock, 2–205
MMG_STD$IOLOCK routine, 2–203 to 2–204
MMG_STD$UNLOCK routine, 2–205 to 2–206
Mount verification routines, 3–29 to 3–30
MT_STD$CHECK_ACCESS routine, 2–207 to 2–208
Mutex
  locking, 1–33, 1–34, 2–209 to 2–210, 2–211 to 2–212
  unlocking, 1–36, 2–213

N

Nonpaged pool
  allocating, 2–47 to 2–49
  deallocating, 2–24 to 2–25
  lookaside list, 2–48

O

OPCOM process
  sending a message to, 2–116, 2–131 to 2–132

P

PCB$L_PID, 2–140
Pending-I/O queue, 2–59 to 2–60, 2–61 to 2–62, 2–90 to 2–92, 2–190
  bypassing, 2–49 to 2–50

Q

QUEUEAST spin lock, 2–33

R

Register dumping routines, 3–31 to 3–32
RELCHAN macro, 1–94
REQCHAN macro, 1–95
REQCOM macro, 1–96
REQPCHAN macro, 1–97
Resource wait mode, 2–48

S

SCH_STD$IOLOCKR routine, 2–209 to 2–210
SCH_STD$IOLOCKW routine, 2–211 to 2–212
SCH_STD$IOUNLOCK routine, 2–213
SS$_ACCVIO, 2–113
SS$_ILLIOFUNC, 2–113
Start I/O routine
  activating, 2–61 to 2–62
  checking for zero length buffer, 2–72, 2–98, 2–119
  transferring control to, 2–90 to 2–92, 2–159 to 2–161
Start-I/O routines
  kernel process, 3–36 to 3–37
  traditional, 3–33 to 3–35
System page-table entry
  allocating permanent, 2–169, 2–172

T

Timeout handling code
  kernel process, 3–40 to 3–41
  traditional, 3–38 to 3–39

U

UCB (unit control block)
  local disk extension, 2–67
UCB$B_DEVCLASS, 2–114
UCB$B_DEVTYPE, 2–114
UCB$B_ONLCNT, 2–65
UCB$L_AFFINITY, 2–160
UCB$L_BCNT, 2–160
UCB$L_BOFF, 2–160
UCB$L_DEVDEPEND, 2–111
UCB$L_IRP, 2–160
UCB$L_SVAPTE, 2–160, 2–169, 2–172
UCB$L_SVPN, 2–168, 2–171
UCB$V_BSY, 2–140
UCB$V_CANCEL, 2–140, 2–160
UCB$V_ERLOGIP, 2–190
UCB$V_LCL_VALID, 2–65
UCB$V_TIMOUT, 2–160
UCB$W_BUFQUO
  in mailbox UCB, 2–132

UCB$W_DEVBUFSIZ, 2–114
  in mailbox UCB, 2–132
UCBLQ_DEVDEPEND, 2–114
Unit delivery routines, 3–42 to 3–44
Unit initialization routines, 3–45 to 3–47

W

WFIKPCH macro, 1–98 to 1–100
WFIRLCH macro, 1–98 to 1–100