Creating an OpenVMS AXP Step 2 Device Driver from a Step 1 Device Driver

Order Number:

August 1993

This manual describes how to create a Step 2 OpenVMS AXP device driver, written in VAX MACRO, to a Step 2 OpenVMS AXP driver, also written in VAX MACRO. A forthcoming book will describe how to write a Step 2 OpenVMS AXP driver in the C programming language.

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

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

© Digital Equipment Corporation 1993.

All Rights Reserved.

This is a draft document. Digital believes that the information in this publication is accurate as of its publication date; such information is subject to change without notice. Digital is not responsible for any errors.

Digital makes no representation that the interconnection of products in the manner described in this publication will not infringe existing or future patent rights. The descriptions contained herein do not imply the granting of any license to make, use or sell products constructed or described.

Redistribution and use of this publication is permitted provided that: (1) any distribution retains this entire copyright notice and comment, and (2) distributions include the following acknowledgement: ''This publication developed by Digital Equipment Corporation.'' in the documentation or other materials provided with the distribution and in all advertising materials mentioning features or use of this software.

The name of Digital Equipment Corporation may not be used to endorse or promote products derived from use of this publication without specific prior written permission.

This publication is provided ''as is'' and without any express or implied warranties, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose.

The following are trademarks of Digital Equipment Corporation: AXP, DEC, DECnet, DECwindows, Digital, HSC, OpenVMS, Q–bus, TURBOchannel, VAX, VAXcluster, VAX DOCUMENT, VAX MACRO, VMScluster, the AXP logo, and the DIGITAL logo.

ZK6321

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

Preface ix

1 Introduction

1.1 Step 2 Features 1–1
1.2 Converting Drivers Written in BLISS 1–3
1.3 Writing Step 2 Drivers In C 1–3

2 Conversion Guidelines

3 Optimizing Step 2 OpenVMS AXP Device Drivers

4 Writing Device-Driver Tables

4.1 Driver Prologue Table 4–1
4.2 Driver Dispatch Table 4–3
4.3 Function Decision Table 4–4
4.3.1 OpenVMS AXP I/O Function Codes 4–6
4.3.1.1 Defining Device-Specific Function Codes 4–9

5 Writing FDT Routines

5.1 Context of Driver FDT Processing 5–2
5.2 Upper-Level FDT Action Routines 5–2
5.2.1 System-Provided Upper-Level FDT Routines 5–4
5.2.2 FDT Exit Paths 5–6
5.3 FDT Routines for System Direct I/O 5–8
5.4 FDT Routines for System Buffered I/O 5–8
5.4.1 Checking Accessibility of the User's Buffer 5–9
5.4.2 Allocating the System Buffer 5–9
5.4.3 Buffered-I/O Postprocessing 5–9

Index

Figures

4–1 Layout of Function Decision Table (FDT) 4–4
5–1 Format of System Buffer for a Buffered-I/O Read Function 5–10

Tables

2–1	Step 2 Upper-Level FDT Action Routines	2–3
2–2	FDT Completion Routines and Macros	2–5
2–3	System-Supplied FDT Support Routines	2–6
2–4	Step 2 Call-Based System Routines and Corresponding Macros	2–10
2–5	Obsolete Step 1 JSB-Based Routines	2–15
4–1	I/O Function Codes	4–6
5–1	System-Provided Upper-Level FDT Action Routines	5–4
5–2	FDT Completion Macros and Associated Routines	5–7

Preface

This manual describes how to convert a Step 1 OpenVMS AXP driver, written in VAX MACRO, to a Step 2 OpenVMS AXP driver, also written in VAX MACRO. A forthcoming book will describe how to write a Step 2 OpenVMS AXP device driver in the C programming language.

Intended Audience

Creating an OpenVMS AXP Step 2 Device Driver from a Step 1 Device Driver is intended for software engineers who must prepare a Step 2 device driver to run on OpenVMS AXP operating system 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 provides an overview of Step 2 device drivers.
- Chapter 2 contains guidelines for converting a Step 1 driver to a Step 2 driver.
- Chapter 3 describes how to improve the performance of a Step 2 device driver.
- Chapter 4 describes how to create driver tables, including the driver prologue table (DPT), driver dispatch table (DDT), and function decision table (FDT).
- Chapter 5 discusses the means by which $QIO system service code dispatches to driver FDT processing to preprocess an I/O function. It explains the role of upper-level FDT action routines, FDT support routines, and FDT completion routines. It also describes the role played by system-supplied FDT action routines and completion routines.

Associated Documents

Creating an OpenVMS AXP Step 2 Device Driver from a Step 1 Device Driver focuses only on those changes that must be made to a Step 1 OpenVMS AXP device driver to produce an equivalent Step 2 OpenVMS AXP device driver. For more detailed information about the macros and routines mentioned in this manual, see the *OpenVMS AXP Step 2 Device Driver Reference Manual*. For basic information about the components of OpenVMS device drivers and OpenVMS requirements for them refer to the following manuals:

- *OpenVMS AXP Device Support: Creating a Step 1 Driver from an OpenVMS VAX Device Driver*

- *OpenVMS VAX Device Support Manual*

Because this manual only addresses the porting to OpenVMS AXP of VAX MACRO coding practices that are typically found in device drivers, readers who need additional information on porting MACRO code, or a detailed description of the MACRO-32 compiler for OpenVMS AXP, should see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*.

Several manuals are available that describe the internals of the OpenVMS AXP operating system and the processes for investigating the types of system failures caused by device drivers. These manuals include:

- *OpenVMS AXP System Dump Analyzer Utility Manual*
- *OpenVMS Delta/XDelta Debugger Manual*
- *VMS for Alpha Platforms: Internals and Data Structures*

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 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.
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
Introduction

OpenVMS AXP Version 1.0 and Version 1.5 provide a documented, but unsupported, mechanism to allow software engineers to create and load device drivers. This mechanism, known as the **Step 1** driver interface, will not be supplied in OpenVMS AXP Version 2.0. At Field Test 3 all Step 1 OpenVMS AXP drivers will stop working. As a result, if you are supplying a driver to run under OpenVMS AXP Version 2.0, it must comply with the **Step 2** driver interfaces described in this manual.

Although Digital intends the Step 2 driver interfaces to allow you to write OpenVMS AXP device drivers in high-level languages, a Step 2 OpenVMS AXP driver can also be written in VAX MACRO. Initially, because most Step 2 OpenVMS AXP drivers were previously Step 1 OpenVMS AXP drivers (and were OpenVMS VAX drivers before that), there will be many VAX MACRO Step 2 OpenVMS AXP drivers.

1.1 Step 2 Features

The primary feature of the Step 2 interface is that it replaces the JSB interfaces used throughout Step 1 drivers with standard call interfaces. The new call interfaces include the following:

- All driver function decision table (FDT) routines are call-based. See Chapter 5 for a description of Step 2 driver FDT processing.
- The following driver routines are call-based:
  - Alternate start-I/O routine
  - Cancel-I/O routine
  - Cloned UCB routine
  - Controller initialization routine
  - Register dumping routine
  - Start-I/O routine
  - Unit delivery routine
  - Unit initialization routine
  - Mount verification routine
  - Channel assign routine
  - Cancel selective routine
- For a device driver that uses the kernel process mechanism, the path from start I/O to request complete remains unchanged. Note, however, that a driver that uses the kernel process mechanism must replace **start**=EXE$KP_

STARTIO in its DDTAB macro invocation with **start**=EXE_STD$KP_ STARTIO.

- In drivers that use the simple fork mechanism, standard call interfaces or JSB interfaces can be used for the code path for start I/O through request complete. For more information, see Chapter 2.
- New call-based system routines, listed in Table 2–4, replace many of the JSB-based routines used by Step 1 drivers. Some JSB-based routines, listed in Table 2–5, have been eliminated. Although the JSB-based routines not listed in that table will continue to be available, a Step 2 driver will often achieve better performance by using the corresponding call-based routines.

The easiest way to convert a Step 1 driver, written in VAX MACRO, to a Step 2 driver is to substitute each JSB to a JSB-based system routine with a macro that calls the new call-based routine. As shown in Table 2–4, many (but not all) new call-based routines have an associated JSB-replacement macro. Each of these macros guarantees the same behavior as the corresponding JSB-based routine (for instance, adopting the same conventions for input, output, and scratch registers), but calling the new call-based routine.

When creating the new call-based system routines, OpenVMS engineering adopted the following conventions:

- The call-based system routine has a different name than its JSB-based counterpart. If x$y is the name of the JSB-based system routine, its call-based counterpart is named x_STD$y. If a JSB-replacement macro exists for x$y, it is named $y. For instance, EXE_STD$FINISHIO is the call-based routine that replaces the JSB-based EXE$FINISHIO. You can replace a JSB to EXE$FINISHIO with the $FINISHIO macro. $FINISHIO issues a standard call to EXE_STD$FINISHIO after loading the AXP processor's argument registers from the general registers used in the conventional JSB to EXE$FINISHIO.
- When using the call-based system routine directly, be aware that its interface may differ from the traditional JSB-based routine.

  Input parameters are usually listed first, specified in the order that corresponds to the register order of the JSB interface input parameters.

  Output parameters are usually listed last, specified in the order that corresponds to the register order of the JSB interface output parameters.

  If a register parameter is both an input and an output to the JSB interface, then it contributes both an input parameter and an output parameter to the new call-based interface.

These conventions serve only as guidelines. In some cases, parameters are dropped or the register order rule is waived if an alternate parameter ordering is more natural. All such interface changes are described in the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

Note

Converting to a Step 2 device driver does not require changes to private driver-level interfaces. Examples of this are the private interfaces between a port and a class driver.

1.2 Converting Drivers Written in BLISS

This manual focuses on converting existing step 1 OpenVMS AXP device drivers, written in VAX MACRO, to step 2 OpenVMS AXP device drivers. However, the call interfaces described are equally available to Step 1 OpenVMS AXP drivers written in BLISS. If you must convert a Step 1 BLISS driver, you proceed by removing the JSB linkages from routine declarations and verifying the specified parameter order for any given routine against that listed in System Routine section of the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

Existing BLISS drivers are likely to have an associated VAX MACRO module that contains the DPTAB, DDTAB, and FUNCTAB declarations, and some routines that were written in VAX MACRO. Currently you must convert these VAX MACRO modules as described in this manual. In the future, it will be possible to replace such a VAX MACRO module with BLISS code.

1.3 Writing Step 2 Drivers In C

OpenVMS AXP Version 2.0 will provide the support necessary to write a device driver in the C programming language. However, this support and the associated documentation are not likely be available early enough to allow sufficient time for the conversion of Step 2 device drivers before Step 1 support is removed from OpenVMS AXP.

2
Conversion Guidelines

This chapter describes the tasks required to convert a Step 1 OpenVMS AXP device driver to a Step 2 OpenVMS AXP device driver. For more details about the macros, system routines, and entry points listed in this chapter, see the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

1. In the driver's DPTAB macro invocation, replace **step= 1** with **step=2**. See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for more information.
2. In the driver's DDTAB macro invocation:
   a. Replace **cancel**=IOC$CANCELIO with **cancel**=IOC_STD$CANCELIO.
   b. Replace **mntver**=IOC$MNTVER with **mntver**=IOC_STD$MNTVER.
   c. Replace **start**=EXE$KP_STARTIO with **start**=EXE_STD$KP_STARTIO. See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for more information about the driver dispatch table (DDT) and the DDTAB macro, respectively.
3. Replace the .JSB_ENTRY MACRO-32 compiler directive at the beginning of each driver entry point with the corresponding macro. Step 2 driver entry point macros include:
   - $DRIVER_ALTSTART_ENTRY **[preserve=<R2,R3,R4,R5>] [,fetch=YES]**
   - $DRIVER_CANCEL_ENTRY **[preserve=<R2,R3,R4>] [,fetch=YES]**
   - $DRIVER_CANCEL_SELECTIVE_ENTRY **[preserve=<>] [,fetch=YES]**
   - $DRIVER_CHANNEL_ASSIGN_ENTRY **[preserve=<>] [,fetch=YES]**
   - $DRIVER_CLONEDUCB_ENTRY **[preserve=<R3>] [,fetch=YES]**
   - $DRIVER_CTRLINIT_ENTRY **[preserve=<R2>] [,fetch=YES]**
   - $DRIVER_FDT_ENTRY **[preserve=<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15>] [,fetch=YES]**
   - $DRIVER_MNTVER_ENTRY **[preserve=<>] [,fetch=YES]**
   - $DRIVER_REGDUMP_ENTRY **[preserve=<R2>] [,fetch=YES]**
   - $DRIVER_START_ENTRY **[preserve=<R2,R4>] [,fetch=YES]**
   - $DRIVER_DELIVER_ENTRY **[preserve=<R2>] [,fetch=YES]**
   - $DRIVER_UNITINIT_ENTRY **[preserve=<R2>] [,fetch=YES]**

   See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for additional information.

4. Ensure that each Step 2 OpenVMS AXP driver routine, mentioned in the previous step, transfers control back to the operating system with a RET instruction, not an RSB instruction.

5. A Step 1 driver contains three or more FUNCTAB macro invocations.

   a. Delete the first FUNCTAB macro—the one that identifies valid I/O function codes—and the function decision table (FDT) label. In their place, insert an FDT_INI macro. The single argument to FDT_INI is the label for the FDT. The label should match the name supplied to the **functb** argument of the DDTAB macro.

      For instance, replace the following code:

```
MK_FUNCTABLE:
        FUNCTAB -
            <NOP,-
             UNLOAD,-
             .
             .
             .
             >
```

      with:

```
        FDT_INI     MK_FUNCTABLE
```

      See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for more information.

   b. Replace the second FUNCTAB—the one that identifies buffered I/O functions —with an FDT_BUF macro. Replacing the word ‘‘FUNCTAB’’ with the word ‘‘FDT_BUF’’ will suffice. For example, replace the following code:

```
        FUNCTAB -
            <NOP,-
             UNLOAD,-
             .
             .
             .
             >
```

      with:

```
        FDT_BUF -
            <NOP,-
             UNLOAD,-
             .
             .
             .
             >
```

      See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for more information.

   c. Replace each subsequent FUNCTAB macro with an FDT_ACT macro. Replacing the word ‘‘FUNCTAB’’ with the word ‘‘FDT_ACT’’ is recommended.

      Because Step 2 OpenVMS AXP driver support replaces all system-supplied upper-level FDT routines with new, callable routines, you must also ensure that each FDT_ACT invocation specifies the correct routine name. Generally, the string ‘‘_STD’’ follows the facility ID and precedes the dollar sign in the routine name.

For example, replace the following code:

```
FUNCTAB EXE$SETMODE, -
    <SETCHAR,-
     SETMODE>
```

with:

```
FDT_ACT EXE_STD$SETMODE, -
    <SETCHAR,-
     SETMODE>
```

Table 2–1 identifies the new Step 2 system-supplied upper-level FDT action routines and the Step 1 routines they replace.

Table 2–1 Step 2 Upper-Level FDT Action Routines

Step 2 FDT Action Routine	Obsolete Step 1 Routine
ACP_STD$ACCESS	ACP$ACCESS
ACP_STD$ACCESSNET	ACP$ACCESSNET
ACP_STD$DEACCESS	ACP$DEACCESS
ACP_STD$MODIFY	ACP$MODIFY
ACP_STD$MOUNT	ACP$MOUNT
ACP_STD$READBLK	ACP$READBLK
ACP_STD$WRITEBLK	ACP$WRITEBLK
EXE$ILLIOFUNC	None
EXE_STD$LCLDSKVALID	EXE$LCLDSKVALID
EXE_STD$MODIFY	EXE$MODIFY
EXE_STD$ONEPARM	EXE$ONEPARM
EXE_STD$READ	EXE$READ
EXE_STD$SENSEMODE	EXE$SENSEMODE
EXE_STD$SETCHAR	EXE$SETCHAR
EXE_STD$SETMODE	EXE$SETMODE
EXE_STD$WRITE	EXE$WRITE
EXE_STD$ZEROPARM	EXE$ZEROPARM
MT_STD$CHECK_ACCESS	MT$CHECK_ACCESS

For more information about these routines, see the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

Note

Step 2 device drivers support only a single upper-level FDT routine per I/O function code. To ensure that no function code is listed more than once, examine a driver's collection of FDT_ACT macro invocations. For those functions that require processing by more than one upper-level FDT action routine, you should provide a new *composite* FDT function which sequentially calls each of the required FDT routines as long as the returned status is successful.

6. If the driver includes a device-specific upper-level FDT action routine, perform the following tasks:

   a. Replace the .JSB_ENTRY MACRO-32 compiler directive used to define the FDT routine entry point with a $DRIVER_FDT_ENTRY macro. (See the *OpenVMS AXP Step 2 Device Driver Reference Manual* .) This macro declares the routine's call entry point and ensures, by default, that all non-scratch registers defined by the OpenVMS Calling Standard are preserved. It also invokes the $FDTARGDEF macro, thus allowing the FDT routine to access its arguments at their standard locations with respect to the AP.

   b. Ensure that the routine does not read R7 to obtain the low-order 6 bits of the $QIO function code parameter, or R8 to obtain the FDT table entry address. It can instead obtain the function code from the IRP and the start of the Step 2 FDT structure from DDT$PS_FDT2. Beware that the Step 2 FDT format differs from the Step 1 format.

   c. Ensure that it issues a RET instruction to return to its caller, the FDT dispatching code in the $QIO system service.

   d. Ensure that it returns SS$_FDT_COMPL warning status to its caller to signify that FDT processing had been completed.

      At some time in FDT processing, a driver calls one of the FDT completion routines listed in Table 2–2. The FDT completion routine returns SS$_FDT_COMPL status in R0 to its caller. This warning status indicates that the IRP is no longer accessible to driver FDT processing. Driver FDT processing must return this value back to FDT dispatching code in the $QIO system service.

7. If driver FDT processing involves specifying an error callback routine as input to one of the Step 1 FDT support routines, EXE$READLOCK_ERR, EXE$MODIFYLOCK_ERR, or EXE$WRITELOCK_ERR, do the following:

   a. Convert the error callback routine to a standard callable routine by using the following entry-point macro:

      $DRIVER_ERRRTN_ENTRY **[preserve=<>] [,fetch=YES]**

      Be sure to specify R2 in the **preserve** mask if the error callback routine EXE$DEANONPAGED, EXE$DEANONPGDSIZ, or some other routine that destroys its contents.

   b. Replace the RSB used by the error callback routine to return to its caller with a RET instruction.

c. Replace the JSB to EXE$READLOCK_ERR, EXE$MODIFYLOCK_ERR, or EXE$WRITELOCK_ERR with the corresponding JSB-replacement macros: $READLOCK_ERR, $MODIFYLOCK_ERR, or $WRITELOCK_ERR.

d. Specify the procedure value of the error callback routine in the **errtn** argument of the macro.

8. Replace the JMP or JSB instructions to Step 1 FDT exit routines with the Step 2 macros (listed in Table 2–2) that call FDT completion routines. As long as **do_ret= YES** (the default) is specified in the macro invocation, the macro simulates Step 1 behavior (that is, it returns control to the FDT dispatching code in the $QIO system service).

Table 2–2 FDT Completion Routines and Macros

Macro	FDT Completion Routine	Obsolete Step 1 FDT Exit Routine
CALL_ABORTIO	EXE_STD$ABORTIO	EXE$ABORTIO
CALL_ALTQUEPKT	EXE_STD$ALTQUEPKT	EXE$ALTQUEPKT
CALL_FINISHIO	EXE_STD$FINISHIO	EXE$FINISHIO
CALL_FINISHIOC	EXE_STD$FINISHIO	EXE$FINISHIOC
CALL_FINISHIO_NOIOST	EXE_STD$FINISHIO	none
CALL_IORSNWAIT	EXE_STD$IORSNWAIT	EXE$IORSNWAIT
CALL_QIOACPPKT	EXE_STD$QIOACPPKT	EXE$QIOACPPKT
CALL_QIODRVPKT	EXE_STD$QIODRVPKT	EXE$QIODRVPKT

If you call an FDT completion routine directly (that is, not by means of a macro), you should note that FDT completion routines:

- Always return to their caller and not to the system service dispatcher.
- Always return the warning status SS$_FDT_COMPL.
- Place the $QIO system service status in a new structure called the FDT_CONTEXT structure.

See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for more information about FDT completion routines and a detailed description of the macros.

9. Replace any JSB instruction to an OpenVMS supplied FDT support routine with the appropriate JSB-replacement macro (see Table 2–3). The macros use the input registers for the corresponding Step 1 FDT support routine as implicit inputs, call the new Step 2 support routine passing the register values in the correct Step 2 parameter order, and restore the output values into the output registers for the corresponding Step 1 routine.

Table 2–3 System-Supplied FDT Support Routines

Macro	FDT Support Routine	Obsolete Step 1 FDT Support Routine
CALL_MODIFYLOCK	EXE_STD$MODIFYLOCK	EXE$MODIFYLOCK
CALL_MODIFYLOCK_ ERR	EXE_STD$MODIFYLOCK	EXE$MODIFYLOCK_ERR
CALL_READCHK	EXE_STD$READCHK	EXE$READCHK
CALL_READCHKR	EXE_STD$READCHK	EXE$READCHKR
CALL_READLOCK	EXE_STD$READLOCK	EXE$READLOCK
CALL_READLOCK_ ERR	EXE_STD$READLOCK	EXE$READLOCK_ERR
CALL_SETATTNAST	COM_STD$SETATTNAST	COM$SETATTNAST
CALL_SETCTRLAST	COM_STD$SETCTRLAST	COM$SETCTRLAST
CALL_WRITECHK	EXE_STD$WRITECHK	EXE$WRITECHK
CALL_WRITECHKR	EXE_STD$WRITECHK	EXE$WRITECHKR
CALL_WRITELOCK	EXE_STD$WRITELOCK	EXE$WRITELOCK
CALL_WRITELOCK_ ERR	EXE_STD$WRITELOCK	EXE$WRITELOCK_ERR

See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for further discussion of system-supplied FDT support routines and details about the macros.

10. Ensure that any driver-supplied FDT support routine checks return status and returns error status to its caller. If the FDT support routine calls an FDT completion routine, the return status will be SS$_FDT_COMPL.

11. No changes are necessary to the driver code paths from start I/O through request complete for drivers that utilize the kernel process mechanism.

    However, because of the new start I/O interface, these drivers will need to specify:

```
DDTAB   -
        STEP     = 2,-
         START   = EXE_STD$KP_STARTIO,-
         KP_STARTIO = driver_startio_routine
```

    instead of:

```
DDTAB   -
        STEP     = 1,-
         START   = EXE$KP_STARTIO,-
         KP_STARTIO = driver_startio_routine
```

    The EXE$KP_STARTIO routine will be made obsolete and will be removed at FT3.

12. Standard call interfaces can be used for the code path from start I/O through request complete in drivers that utilize the simple fork mechanism. This code path can be converted to the standard call interfaces in drivers written in Macro-32 by following these steps:

    a. Use the $DRIVER_START_ENTRY and $DRIVER_ALTSTART_ENTRY macros to define the driver's start I/O and appropriate alternate start I/O routines.

b. Use the ENVIRONMENT=CALL keyword parameter on the FORK, FORK_ROUTINE, FORK_WAIT, IOFORK, REQCOM, REQCHAN, REQPCHAN, WFIKPCH, and WFIRLCH macros. A detailed description of the Start I/O to REQCOM Conversion Implications For Step 1 Macro-32 Drivers changes to these macros can be found in the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

c. Use the FORK_ROUTINE macro (with ENVIRONMENT=CALL), the .CALL_ENTRY directive, or the .ENTRY directive instead of .JSB_ENTRY to define the entry points for driver fork, channel grant, resume from interrupt, and interrupt timeout routines. A detailed description of these driver interfaces can be found in the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

d. Use care in accessing the actual parameters to the above routines that are now passed according to the calling standard. In Macro-32 source code these parameters are accessed using AP offsets. For example, the FORK_ROUTINE macro defines the offsets FORKARG$_FR3, FORKARG$_FR4, and FORKARG$_FKB for use in accessing fork routine parameters. Note also that some of the above macros will copy the standard call parameters into the input registers expected by the old JSB interfaces.

e. Use the RET instruction instead of the RSB instruction to return from all of the above standard call interface routines.

f. Use the scratch registers as defined by the the calling standard. Some of the old JSB interface routines were allowed to scratch registers R2 through R5 which are not in the scratch register set as defined by the calling standard. Also, the calling standard allows R0 and R1 to be scratched by a called routine while some of the JSB interface routines preserve R0 or R1.

g. Use the following code sequence to invoke the driver interrupt resume routine from the driver interrupt service routine:

```
PUSHL     R5                    ;P3 = UCB from R5
PUSHL     UCB$Q_FR4(R5)         ;P2 = FR4 (32-bits)
PUSHL     UCB$Q_FR3(R5)         ;P1 = FR3 (32-bits)
CALLS     #3,@UCB$L_FPC(R5)     ;call driver routine
```

as a replacement for:

```
MOVL      UCB$Q_FR3(R5),R3      ;R3 = FR3 (32-bits)
MOVL      UCB$Q_FR4(R5),R4      ;R4 = FR4 (32-bits)
JSB       @UCB$L_FPC(R5)        ;call driver routine
```

If your driver needs to preserve the full 64-bits of its FR3 or FR4 parameters, then it can use the following:

```
MOVX      UCB$Q_FR3(R5),R16     ;R16 = FR3 (64-bits)
MOVX      UCB$Q_FR4(R5),R17     ;R17 = FR4 (64-bits)
PUSHL     R5                    ;P3 = UCB from R5
PUSHL     R17                   ;P2 = 64-bits of R17
PUSHL     R16                   ;P1 = 64-bits of R16
CALLS     #3,@UCB$L_FPC(R5)     ;call driver routine
```

Note that although the above is fairly opaque source, it results in code that is just as efficient as that produced by the first example.

Note

For more details about this code sequence, see the description of the FORK ROUTINE interface in Chapter 3 of the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

The called routine can obtain 64-bit parameter values by declaring its entry point using the FORK_ROUTINE macro or the WFIKPCH macro.

h. Examine the inter-routine branches between the above routines and other routines in the same modules. All these routines need to be changed to standard call interfaces as well.

i. Examine the relevant source if you encounter any of the following Macro-32 compiler diagnostic messages:

```
%AMAC-E-ILLRSBCAL, illegal RSB in CALL_ENTRY routine

%AMAC-I-BRINTOCAL, branch into CALL_ENTRY routine from
            JSB_ENTRY

%AMAC-I-JSBHOME, arglist use in JSB entry requires homed
            arglist in caller

%AMAC-I-RETINJSB, RET in JSB_ENTRY, with non-scratch
            registers
```

These messages are likely to result from a .JSB_ENTRY routine that needs to be converted to a standard call entry. Note, however, that in some cases the last three diagnostics above may be encountered under acceptable circumstances. If this is so, you should document the reasons and consider disabling the diagnostic message by bracketing the smallest possible section of relevant code with:

```
.DSABL  FLAGGING
.
.
.ENABL  FLAGGING
```

In particular, the usage of a RET from a JSB entry routine may be allowable in a step 2 driver in the context of complex FDT routines. However, you will be able to improve the performance of the code path if you can alter the source to avoid the need for a RET in a JSB routine.

13. For Drivers Continuing To Use JSB Based Fork Routines, the code path from start I/O through request complete in some existing drivers written in Macro-32 may be difficult and error prone to convert to the standard call interfaces. This is likely to apply to complex drivers that make extensive use of branches between routines within the same module.

    Such drivers can choose to continue to use the traditional JSB interfaces for their start I/O through request complete code path. These drivers will need to use the DDTAB JSB_START parameter to specify their start I/O entry point:

```
DDTAB   -
        STEP    = 2,-
        JSB_START = driver_startio_routine
```

    instead of:

```
DDTAB   -
        STEP    = 1,-
        START   = driver_startio_routine
```

By doing so, the IOC$START_C2J CALL-to-JSB jacket routine is actually used as the start I/O entry. The IOC$START_C2J routine invokes the routine specified by the JSB_START parameter. Thus, the above is equivalent to:

```
DDTAB   -
        STEP    = 2,-
        START   = IOC$START_C2J,-
        JSB_START = driver_startio_routine
```

A similar approach can also be used for the alternate start I/O entry point. The DDTAB JSB_ALTSTART parameter is used to specify the alternate start I/O entry:

```
DDTAB   -
        STEP     = 2,-
         JSB_ALTSTART = driver_altstart_routine
```

instead of:

```
DDTAB   -
        STEP     = 1,-
         ALTSTART = driver_altstart_routine
```

The above is equivalent to:

```
DDTAB   -
        STEP    = 2,-
        ALTSTART = IOC$ALTSTART_C2J,-
        JSB_ALTSTART = driver_altstart_routine
```

The performance cost of this approach is one additional level of routine call to dispatch an IRP to the driver's start I/O routine or alternate start I/O routine.

The driver macros FORK, FORK_ROUTINE, FORK_WAIT, IOFORK, RELCHAN, REQCOM, REQCHAN, REQPCHAN, WFIKPCH, and WFIRLCH can continue to be used in the same manner as in a Step 1 driver. By default these macros will assume a JSB fork environment. Note, however, that these routines may expand to calls to new routines (for example, WFIKPCH will call IOC_STD$PRIMITIVE_WFIKPCH) but all the implicit inputs and outputs are preserved.

14. Replace any JSB to the routines listed in Table 2–4 with the appropriate macro, if one is provided. Macros are available to help you convert most JSBs to system routines for which the Step 1 JSB-based interface is not provided at Step 2. Macros are also provided for many system routines for which the JSB-based interface continues to exist at Step 2, alongside a new call-based interface. These macros guarantee the most direct control transfer to the system routine. You can optimize performance of the macro by following the recommendations listed in Chapter 3.

    If the interface provided by the JSB-replacement macro differs from the original JSB interface, the macro generates a compile-time warning. Certain macros ensure compatibility with the original JSB interface by saving R0, R1, or both. These macros provide an argument that allows you to specify that these registers not be saved. See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for a detailed description of the macros.

Most JSB-based routines corresponding to the new call-based routines listed in Table 2–4 continue to be available to step 2 drivers. However, in many cases, the new call-based interface routine provides better performance than the JSB-based interfaces. If you intend to call a call-based system routine directly (without using a macro), check the ‘‘Notes for Converting Step 1 Drivers’’ section of the routine’s description in *OpenVMS AXP Step 2 Device Driver Reference Manual* to verify the routine interface.

Table 2–4 Step 2 Call-Based System Routines and Corresponding Macros

Call Interface	Macro	Interface Warning	Save R0/R1
ACP_STD$ACCESS[1]	None	—	—
ACP_STD$ACCESSNET[1]	None	—	—
ACP_STD$DEACCESS[1]	None	—	—
ACP_STD$MODIFY[1]	None	—	—
ACP_STD$MOUNT[1]	None	—	—
ACP_STD$READBLK[1]	None	—	—
ACP_STD$WRITEBLK[1]	None	—	—
COM_STD$DELATTNAST	CALL_DELATTNAST	No	No
COM_STD$DELATTNASTP	CALL_DELATTNASTP	No	No
COM_STD$DELCTRLAST	CALL_DELCTRLAST	No	No
COM_STD$DELCTRLASTP	CALL_DELCTRLASTP	No	No
COM_STD$DRVDEALMEM	CALL_DRVDEALMEM	No	No
COM_STD$FLUSHATTNS	CALL_FLUSHATTNS	No	No
COM_STD$FLUSHCTRLS	CALL_FLUSHCTRLS	No	No
COM_STD$POST	CALL_POST	No	No
COM_STD$POST_NOCNT	CALL_POST_NOCNT	No	No
COM_STD$SETATTNAST[1]	CALL_SETATTNAST	No	No
COM_STD$SETCTRLAST[1]	CALL_SETCTRLAST	No	No
ERL_STD$ALLOCEMB	CALL_ALLOCEMB	No	No
ERL_STD$DEVICEATTN	CALL_DEVICEATTN	No	No
ERL_STD$DEVICERR	CALL_DEVICERR	No	No
ERL_STD$DEVICTMO	CALL_DEVICTMO	No	No
ERL_STD$RELEASEMB	CALL_RELEASEMB	No	No
EXE_STD$ABORTIO[1]	CALL_ABORTIO	No	No
EXE_STD$ALLOCBUF	CALL_ALLOCBUF	No	No
EXE_STD$ALLOCCEB	None	—	—
EXE_STD$ALLOCIRP	CALL_ALLOCIRP	No	No
EXE_STD$ALLOCJIB	None	—	—
EXE_STD$ALLOCPCB	None	—	—
EXE_STD$ALLOCTQE	None	—	—

[1]The JSB-based Step 1 routine corresponding to this call-based Step 2 routine is obsolete and will be removed at Field Test 3 from the OpenVMS AXP operating system.

(continued on next page)

Table 2–4 (Cont.) Step 2 Call-Based System Routines and Corresponding Macros

Call Interface	Macro	Interface Warning	Save R0/R1
EXE_STD$ALONONPAGED	None	—	—
EXE_STD$ALOPHYCNTG	None	—	—
EXE_STD$ALTQUEPKT	CALL_ALTQUEPKT	No	No
EXE_STD$CARRIAGE	CALL_CARRIAGE	No	No
EXE_STD$CHKPRO_INT	None	—	—
EXE_STD$CHKCREACCES	CALL_CHKCREACCES	No	R1
EXE_STD$CHKDELACCES	CALL_CHKDELACCES	No	R1
EXE_STD$CHKEXEACCES	CALL_CHKEXEACCES	No	R1
EXE_STD$CHKLOGACCES	CALL_CHKLOGACCES	No	R1
EXE_STD$CHKPHYACCES	CALL_CHKPHYACCES	No	R1
EXE_STD$CHKRDACCES	CALL_CHKRDACCES	No	R1
EXE_STD$CHKWRTACCES	CALL_CHKWRTACCES	No	R1
EXE_STD$CREDIT_BYTCNT_BYTLM	None	—	—
EXE_STD$CVT_IPID_TO_EPID	None	—	—
EXE_STD$CVT_IPID_TO_PCB	None	—	—
EXE_STD$DEANONPAGED	None	—	—
EXE_STD$DEANONPGDSIZ	None	—	—
EXE_STD$DEBIT_BYTCNT	None	—	—
EXE_STD$DEBIT_BYTCNT_ALO	None	—	—
EXE_STD$DEBIT_BYTCNT_BYTLM_NW	None	—	—
EXE_STD$FINISHIO[1]	CALL_FINISHIO, CALL_FINISHIOC	No	No
EXE_STD$INSERT_IRP	CALL_INSERT_IRP	No	No
EXE_STD$INSIOQ	CALL_INSIOQ	No	No
EXE_STD$INSIOQC	CALL_INSIOQC	No	No
EXE_STD$INSTIMQ	None	—	—
EXE_STD$IORSNWAIT[1]	CALL_IORSNWAIT	No	No
EXE_STD$KP_STARTIO[1]	None	—	—
EXE_STD$LCLDSKVALID[1]	None	—	—
EXE_STD$MAXACMODE	None	—	—
EXE_STD$MNTVERSIO	CALL_MNTVERSIO	No	No
EXE_STD$MODIFY[1]	None	—	—
EXE_STD$MODIFYLOCK[1]	CALL_MODIFYLOCK CALL_MODIFYLOCK_ERR	No Yes	No No
EXE_STD$MOUNT_VER	CALL_MOUNT_VER	No	R0 and R1
EXE_STD$ONEPARM[1]	None	—	—

[1]The JSB-based Step 1 routine corresponding to this call-based Step 2 routine is obsolete and will be removed at Field Test 3 from the OpenVMS AXP operating system.

(continued on next page)

Table 2–4 (Cont.) Step 2 Call-Based System Routines and Corresponding Macros

Call Interface	Macro	Interface Warning	Save R0/R1
EXE_STD$OUTZSTRING	None	—	—
EXE_STD$PRIMITIVE_FORK	FORK[2]	No	No
EXE_STD$PRIMITIVE_FORK_WAIT	FORK_WAIT[2]	No	No
EXE_STD$PROBER	None	—	—
EXE_STD$PROBER_DSC	None	—	—
EXE_STD$QIOACPPKT[1]	CALL_QIOACPPKT	No	No
EXE_STD$QIODRVPKT[1]	CALL_QIODRVPKT	No	No
EXE_STD$QXQPPKT[1]	CALL_QQXQPPKT	No	No
EXE_STD$QUEUE_FORK	None	—	—
EXE_STD$READ[1]	None	—	—
EXE_STD$READCHK [1]	CALL_READCHK CALL_READCHKR	No No	No No
EXE_STD$READLOCK[1]	CALL_READLOCK CALL_READLOCK_ERR	No Yes	No No
EXE_STD$RMVTIMQ	None	—	—
EXE_STD$SENSEMODE[1]	None	—	—
EXE_STD$SETCHAR[1]	None	—	—
EXE_STD$SETMODE[1]	None	—	—
EXE_STD$SNDEVMSG	CALL_SNDEVMSG	No	No
EXE_STD$SNGLEQUOTA_LONG	None	—	—
EXE_STD$SYNCH_LOOP	None	—	—
EXE_STD$WRITE[1]	None	—	—
EXE_STD$WRITECHK[1]	CALL_WRITECHK CALL_WRITECHKR	No No	No No
EXE_STD$WRITELOCK[1]	CALL_WRITELOCK CALL_WRITELOCK_ERR	No Yes	No No
EXE_STD$WRTMAILBOX	CALL_WRTMAILBOX	No	No
EXE_STD$ZEROPARM[1]	None	—	—
IOC_STD$ALLOC_CNT_RES	TBS	TBS	No
IOC_STD$ALTREQCOM	CALL_ALTREQCOM	No	No
IOC_STD$BROADCAST	CALL_BROADCAST	No	R1
IOC_STD$CANCELIO	CALL_CANCELIO	No	R0 and R1
IOC_STD$CLONE_UCB[1]	CALL_CLONE_UCB	Yes	No
IOC_STD$COPY_UCB	CALL_COPY_UCB	No	No
IOC_STD$CREDIT_UCB	CALL_CREDIT_UCB	No	No

[1]The JSB-based Step 1 routine corresponding to this call-based Step 2 routine is obsolete and will be removed at Field Test 3 from the OpenVMS AXP operating system.

[2]The standard call interface version of the routine is used by the macro if the ENVIRONMENT=CALL parameter is specified.

(continued on next page)

Table 2–4 (Cont.) Step 2 Call-Based System Routines and Corresponding Macros

Call Interface	Macro	Interface Warning	Save R0/R1
IOC_STD$CVTLOGPHY	CALL_CVTLOGPHY	No	No
IOC_STD$CVT_DEVNAM	CALL_CVT_DEVNAM	No	No
IOC_STD$DELETE_UCB	CALL_DELETE_UCB	No	No
IOC_STD$DIAGBUFILL	CALL_DIAGBUFILL	No	No
IOC_STD$FILSPT	CALL_FILSPT	No	No
IOC_STD$GETBYTE	CALL_GETBYTE	No	No
IOC_STD$INITBUFWIND	CALL_INITBUFWIND	No	No
IOC_STD$INITIATE	CALL_INITIATE	No	No
IOC_STD$LINK_UCB[1]	CALL_LINK_UCB	Yes	No
IOC_STD$MAPVBLK	CALL_MAPVBLK	No	No
IOC_STD$MNTVER	CALL_MNTVER	No	No
IOC_STD$MOVFRUSER	CALL_MOVFRUSER	No	No
IOC_STD$MOVFRUSER2	CALL_MOVFRUSER2	No	No
IOC_STD$MOVTOUSER	CALL_MOVTOUSER	No	No
IOC_STD$MOVTOUSER2	CALL_MOVTOUSER2	No	No
IOC_STD$PARSDEVNAM	CALL_PARSDEVNAM	No	No
IOC_STD$POST_IRP	CALL_POST_IRP	No	No
IOC_STD$PRIMITIVE_REQCHANH[1]	REQCHAN	No	No
IOC_STD$PRIMITIVE_REQCHANL[1]	REQCHAN	No	No
IOC_STD$PRIMITIVE_WFIKPCH[1]	WFIKPCH	No	No
IOC_STD$PRIMITIVE_WFIRLCH[1]	WFIRLCH	No	No
IOC_STD$PTETOPFN	CALL_PTETOPFN	No	No
IOC_STD$QNXTSEG1	CALL_QNXTSEG1	No	No
IOC_STD$RELCHAN	RELCHAN	No	No
IOC_STD$REQCOM	REQCOM	No	No
IOC_STD$SEARCHDEV	CALL_SEARCHDEV	No	No
IOC_STD$SEARCHINT	CALL_SEARCHINT	No	No
IOC_STD$SEVER_UCB	CALL_SEVER_UCB	No	No
IOC_STD$SIMREQCOM	CALL_SIMREQCOM	No	No
IOC_STD$THREADCRB	CALL_THREADCRB	No	R0
LDR_STD$ALLOC_PT	None	—	—
LDR_STD$DEALLOC_PT	None	—	—
LNM_STD$DELETE_LNMB	None	—	–
LNM_STD$LOCKW	None	—	—
LNM_STD$UNLOCK	None	—	—
MMG_STD$ALLOC_PFN	None	—	—

[1]The JSB-based Step 1 routine corresponding to this call-based Step 2 routine is obsolete and will be removed at Field Test 3 from the OpenVMS AXP operating system.

(continued on next page)

Table 2–4 (Cont.) Step 2 Call-Based System Routines and Corresponding Macros

Call Interface	Macro	Interface Warning	Save R0/R1
MMG_STD$DALLOC_PFN	None	—	—
MMG_STD$IOLOCK	CALL_IOLOCK	No	No
MMG_STD$SVAPTECHK	None	—	—
MMG_STD$UNLOCK	CALL_UNLOCK	No	No
MT_STD$CHECK_ACCESS[1]	None	—	—
PMS_STD$ABORT_RQ	None	—	—
PMS_STD$START_RQ	None	—	—
SCH_STD$CLREF	None	—	—
SCH_STD$POSTEF	None	—	—
SCH_STD$IOLOCKR	CALL_IOLOCKR	No	R1
SCH_STD$IOLOCKW	CALL_IOLOCKW	No	No
SCH_STD$IOUNLOCK	CALL_IOUNLOCK	No	No
SCH_STD$LOCKW	None	—	—
SCH_STD$LOCKWEXEC	None	—	—
SCH_STD$QAST	None	—	—
SCH_STD$RAVAIL	None	—	—
SCH_STD$LOCKWEXEC_QUAD	None	—	—
SCH_STD$UNLOCK	None	—	—
SCH_STD$UNLOCKEXEC	None	—	—
SCH_STD$UNLOCKEXEC_QUAD	None	—	—
SCH_STD$WAKE	None	—	—
SMP_STD$ACQNOIPL	None	—	—
SMP_STD$ACQUIRE	None	—	—
SMP_STD$ACQUIREL	None	—	—
SMP_STD$RELEASE	None	—	—
SMP_STD$RELEASEL	None	—	—
SMP_STD$RESTORE	None	—	—
SMP_STD$RESTOREL	None	—	—

[1]The JSB-based Step 1 routine corresponding to this call-based Step 2 routine is obsolete and will be removed at Field Test 3 from the OpenVMS AXP operating system.

15. Replace JSBs to the routines listed in Table 2–5 with the appropriate macro, if one is provided. If the interface provided by the JSB-replacement macro differs from the original JSB interface, the macro generates a compile-time warning. See *OpenVMS AXP Step 2 Device Driver Reference Manual* for a detailed description of the macros.

    If you must call a call-based system routine directly (without using a macro), check the "Notes for Converting Step 1 Drivers" section of the routine's description in *OpenVMS AXP Step 2 Device Driver Reference Manual* to verify the routine interface.

Note

At a later stage in the development of the Step 2 driver, OpenVMS AXP engineering will provide an alternate version of the SYS$BASE_IMAGE.EXE. By linking against this version of the system base image, you can ensure that your driver issues no JSBs to obsolete JSB-based routines.

Table 2–5 Obsolete Step 1 JSB-Based Routines

Obsolete Routine	Replacement Routine	Macro
ACP$ACCESS	ACP_STD$ACCESS	—
ACP$ACCESSNET	ACP_STD$ACCESSNET	—
ACP$DEACCESS	ACP_STD$DEACCESS	—
ACP$MODIFY	ACP_STD$MODIFY	—
ACP$MOUNT	ACP_STD$MOUNT	—
ACP$READBLK	ACP_STD$READBLK	—
ACP$WRITEBLK	ACP_STD$WRITEBLK	—
COM$SETATTNAST	COM_STD$SETATTNAST	CALL_SETATTNAST
COM$SETCTRLAST	COM_STD$SETCTRLAST	CALL_SETCTRLAST
EXE$ABORTIO	EXE_STD$ABORTIO	CALL_ABORTIO
EXE$FINISHIO	EXE_STD$FINISHIO	CALL_FINISHIO
EXE$FINISHIOC	EXE_STD$FINISHIO	CALL_FINISHIOC
EXE$IORSNWAIT	EXE_STD$IORSNWAIT	CALL_IORSNWAIT
EXE$KP_STARTIO	EXE_STD$KP_STARTIO	—
EXE$LCLDSKVALID	EXE_STD$LCLDSKVALID	—
EXE$MODIFY	EXE_STD$MODIFY	—
EXE$MODIFYLOCK	EXE_STD$MODIFYLOCK	CALL_MODIFYLOCK
EXE$MODIFYLOCK_ERR	EXE_STD$MODIFYLOCK	CALL_MODIFYLOCK_ERR
EXE$ONEPARM	EXE_STD$ONEPARM	—
EXE$QIOACPPKT	EXE_STD$QIOACPPKT	CALL_QIOACPPKT
EXE$QIODRVPKT	EXE_STD$QIODRVPKT	CALL_QIODRVPKT
EXE$QIORETURN	None[1]	—
EXE$READ	EXE_STD$READ	—
EXE$READCHK	EXE_STD$READCHK	CALL_READCHK
EXE$READCHKR	EXE_STD$READCHK	CALL_READCHKR
EXE$READLOCK	EXE_STD$READLOCK	CALL_READLOCK
EXE$READLOCK_ERR	EXE_STD$READLOCK_ERR	CALL_READLOCK_ERR

[1]If your driver issues a JSB or JMP instruction to EXE$QIORETURN, you must replace the JSB or JMP with code that:

- Releases the device lock if held. EXE$QIORETURN contains code that unconditionally releases the device lock.
- Places SS$_FDT_COMPL status in R0 before returning to its caller. Because the final system service status in the FDT_CONTEXT structure is SS$_NORMAL by default, the driver does nothing special to deliver this status to the $QIO caller.

(continued on next page)

Table 2–5 (Cont.) Obsolete Step 1 JSB-Based Routines

Obsolete Routine	Replacement Routine	Macro
EXE$SENSEMODE	EXE_STD$SENSEMODE	—
EXE$SETCHAR	EXE_STD$SETCHAR	—
EXE$SETMODE	EXE_STD$SETMODE	—
EXE$WRITE	EXE_STD$WRITE	—
EXE$WRITECHK	EXE_STD$WRITECHK	CALL_WRITECHK
EXE$WRITECHKR	EXE_STD$WRITECHK	CALL_WRITECHKR
EXE$WRITELOCK	EXE_STD$WRITELOCK	CALL_WRITELOCK
EXE$WRITELOCK_ERR	EXE_STD$WRITELOCK	CALL_WRITELOCK_ERR
EXE$ZEROPARM	EXE_STD$ZEROPARM	—
IOC$CLONE_UCB	IOC_STD$CLONE_UCB	CALL_CLONE_UCB
IOC$LINK_UCB	IOC_STD$LINK_UCB	CALL_LINK_UCB
MT$CHECK_ACCESS	MT_STD$CHECK_ACCESS	—

16. Note that device interrupt timeouts are handled differently for Step 2 drivers. For Step 1 drivers the UCB$L_FPC cell in the device Unit Control Block contained the procedure value of the routine that served as both the resume from interrupt routine and the interrupt timeout routine. These two routine are now separate. The new UCB cell UCB$PS_TOUTROUT is used for the procedure value of the interrupt timeout routine.

    These changes are transparent to code that uses the WFIKPCH or WFIRLCH macros, or calls the IOC$PRIMITIVE_WFIKPCH or IOC$PRIMITIVE_ WFIRLCH routines. However, code that manually sets the UCB$V_TIM bit in UCB$L_STS will now need to place the timeout routine procedure value into UCB$PS_TOUTROUT, instead of in UCB$L_FPC. For more information, see the specific routine descriptions in the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

3

Optimizing Step 2 OpenVMS AXP Device Drivers

Once you have successfully converted a Step 1 OpenVMS AXP device driver to an OpenVMS AXP Step 2 device driver, you may want to improve the driver's performance. It is wisest, however, to first ensure that the driver has been converted correctly and has been adequately tested. The JSB-replacement macros and driver-entry-point macros were designed to facilitate conversion. Fine-tuning their operation requires a scrupulous examination of the codepaths into which they lead, as well as some serious probing of code that precedes and follows their invocation.

As mentioned in Chapter 2, you should always use a JSB-replacement macro in instances where the Step 1 JSB-based interface to a system routine and a new call-based interface coexist. Either the JSB entry point or the call entry point may be a jacket. Using the macro guarantees that the jacket call is avoided.

You can replace a JSB to a JSB-based system routine in a Step 1 driver with a JSB-replacement macro. The JSB-replacement macro accepts the same input registers and provides the same output registers as the corresponding Step 1 JSB-based routine. In some cases, you can specify that R0, R1, or both R0 and R1 not be saved if the driver does not need them preserved. (These macros have an argument named **save_r0**, **save_r1**, or, **save_r0r1**.) Eliminating unneeded 64-bit saves of these registers is a performance gain.

Similarly, you can adapt a driver's use of the driver-entry-point macros such that it more closely affects the behavior of driver routines.

Each driver-entry-point macro, by default, initializes the general-purpose registers a Step 1 driver routine expects as input. At the very least, this practice requires a series register-to-register loads, plus, by virtue of the default behavior of the MACRO-32 compiler (which automatically preserves any register an entry point modifies), a set of 64-bit register save and restore operations. If the execution path initiated at a driver entry point does not use one or more of the registers defined as Step 1 input registers, you might consider specifying **fetch=NO** and explicitly loading the registers it does use.

Each driver-entry-point macro, by default, preserves a set of registers across a call. The MACRO-32 compiler, by default, preserves those registers the routine explicitly modifies (but not those implicitly modified by a system routine or driver-specific routine it calls). Here, too, if the execution path initiated at a driver entry point does not use one or more of the registers defined as Step 1 input registers, you might consider removing them from the **preserve** mask. Before doing so, carefully examine the chain of execution that proceeds from the entry point to ensure that some inconspicuous code path does not alter a register you would like to remove from the mask.

For instance, the $DRIVER_FDT_ENTRY macro specifies, by default, that registers R2 through R15 be preserved. For certain FDT entry points, you can specify a much smaller set of registers — **preserve=<R2,R9,R10,R11>** is usually sufficient. (These registers are allowed to be scratched by Step 1 FDT routines.)

You can safely follow this recommendation only if the FDT processing initiated by the upper-level FDT action routine avoids the situation in which a call initiated by a JSB instruction is concluded by a RET instruction instead of an RSB (a situation for which the MACRO-32 compiler generates a warning). A "RET under JSB" can occur in FDT processing if the upper-level FDT routine issues a JSB to an FDT support routine that invokes an FDT completion macro (see Table 2–2 without specifying **do_ret=NO**. The additional RET instruction generated by a default invocation of the macro would return control back to FDT dispatching code in the $QIO system service, and risks the destruction of register context required by that code.

4
Writing Device-Driver Tables

Every device driver declares three static tables that describe the device and driver:

- **Driver prologue table (DPT)**—Describes the device type, driver name, and fields in the I/O database to be initialized during driver loading and reloading.
- **Driver dispatch table (DDT)**—Lists some of the driver's entry points to which the operating system transfers control. The controller request block and function decision table list other entry points.
- **Function decision table (FDT)**—Provides the names of action routines for I/O functions the driver supports and indicates which of those functions it supports by using an intermediate system buffer.

The operating system provides macros that drivers can invoke to create these tables.

4.1 Driver Prologue Table

The driver prologue table (DPT) is the first part of every device driver. This table, along with parameters to the System Management utility (SYSMAN) command that request driver loading, describes the driver to the driver-loading procedure. The driver-loading procedure performs the following tasks:

- Further initializes the DPT
- Creates data structures for the new devices in the I/O database
- Calls the OpenVMS executive loader to compute the size of the driver and load it into nonpaged system memory
- Links the new DPT into a list of all DPTs known to the system (IOC$GL_ DPTLIST).

Device drivers can pass data-structure initialization information to the driver-loading procedure through values stored in the DPT. In addition, the driver-loading procedure initializes some fields within the I/O database using information from its own tables.

To create a DPT, a driver invokes the DPTAB macro, as described in the *OpenVMS AXP Step 2 Device Driver Reference Manual*. The DPTAB macro generates a driver prologue table (DPT) in a program section called $$$105_ PROLOGUE.

The DPTAB macro requires the following information:

- An indication that the driver conforms to the coding practices for a Step 2 OpenVMS AXP device driver by supplying **step=2** in the **step** argument.

- An indication that the driver has been written to run in a symmetric multiprocessing system. All OpenVMS AXP drivers must specify **smp=YES**.
- Code identifying the device by its adapter type in the **adapter** argument. 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.
- Size of the unit control block (UCB) in the **ucbsize** argument.

The DPTAB also allows you to specify the following information, if applicable to the device driver:

- Whether the driver needs a permanently allocated system page
- Whether the driver needs controller register access mailboxes (CRAMs) to be preallocated and associated with the unit control block (UCB) or interrupt dispatch block (IDB).
- Maximum number of units supported by the driver (default is 8)
- Number of UCBs to be created when the driver is loaded by means of the System Management (SYSMAN) utility's autoconfiguration facility and the address of a unit delivery routine to be called by that facility

A driver typically follows the DPTAB macro invocation with several instances of the DPT_STORE macro and the DPT_STORE_ISR macro, which allow the driver to communicate its initialization needs to the driver-loading procedure and reloading procedures. A driver uses these macros in the following manner:

- A driver invokes the DPT_STORE macro once, specifying the **INIT** keyword, to automatically generate a driver I/O database initialization routine. This routine, which executes when the driver is first loaded into the system, initializes those fields specified by the series of DPT_STORE macros that intervene between the DPT_STORE **INIT** and DPT_STORE **REINIT** declarations.

  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

- A driver invokes the DPT_STORE macro once, specifying the **REINIT** keyword, to automatically generate a driver I/O database reinitialization routine. This routine, which executes when the driver is first loaded into the system, and whenever the driver is reloaded, initializes those fields specified by the series of DPT_STORE and DPT_STORE_ISR macros that intervene between the DPT_STORE **REINIT** and DPT_STORE **END** declarations.

  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

- A driver invokes the DPT_STORE macro once, specifying the **END** keyword, to denote the end of the reinitialization section.

For additional information on how to use the DPTAB, DPT_STORE, and DPT_STORE_ISR macros, plus an example of their usage, see the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

4.2 Driver Dispatch Table

The driver dispatch table (DDT) identifies those driver routines that the operating system calls to process I/O requests. Every driver must create a DDT.

The routines listed in the DDT can reside in the driver module or in a system module. Device-dependent routines are normally located in the driver module. For more information about the system device-independent routines that can be specified, see the Systems Routines section of the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

The driver creates a DDT by invoking the macro DDTAB. The DDTAB macro generates a DDT named **devnam**$DDT, according to the value you supply in **devnam** macro argument. It invokes the DRIVER_DATA macro to place the DDT within the the driver's data program section ($$$110_DATA).

When the driver is loaded into memory, OpenVMS writes the address of its DDT into DDB$L_DDT.

You must specify the address of the driver's function decision table (FDT) in the **functb** argument of the DDTAB macro. Several optional arguments allow you to specify the names of the following routines, if applicable:

- Start-I/O routine
- Controller initialization routine
- Cancel-I/O routine
- Register dumping routine

- Unit initialization routine
- Alternate start-I/O routine
- Mount verification routine
- Cloned UCB routine
- Start-I/O routine for a driver employing the kernel process services

In addition, the DPTAB macro allows you to specify the length of any diagnostic buffer or error message buffer.

See the description of the DDTAB macro in the *OpenVMS AXP Step 2 Device Driver Reference Manual* for additional information.

4.3 Function Decision Table

A function decision table (FDT) is a structure within a driver that consists of two substructures:

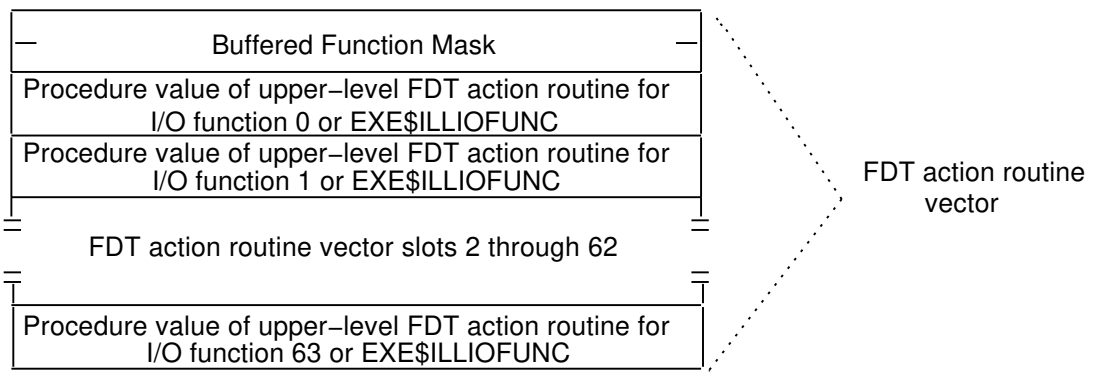
- A quadword bitmap known as the buffered function mask
- A 64-element array of longwords known as the FDT action routine vector

Each bit in the **buffered function mask** represents an I/O function that is serviced by the driver by means of an intermediate system buffer.

Each 32-bit slot of the **FDT action routine vector** corresponds to the symbolic value of an I/O function defined by the $IODEF macro in SYS$LIBRARY:STARLET.MLB. Those vector slots that relate to functions serviced by a driver contain the procedure value of an upper-level FDT action routine that initiates driver-specific preprocessing of the function. Those slots that represent functions the driver does not support contain the procedure value of a system upper-level FDT action routine that processes illegal I/O functions (EXE$ILLIOFUNC). When a $QIO is issued to a device driver, specifying an I/O function code the driver does not support, EXE$ILLIOFUNC executes, calling the FDT completion routine EXE_STD$ABORTIO. EXE_STD$ABORTIO terminates the I/O request and passes SS$_ILLIOFUNC status back to the $QIO caller.

Figure 4–1 depicts the layout of the FDT.

Figure 4–1 Layout of Function Decision Table (FDT)



A driver uses the following three macros in sequence to create and initialize its FDT:

- FDT_INI—Creates, labels, and initializes an FDT
- FDT_BUF—Creates the buffered function mask within an FDT
- FDT_ACT—Initializes one or more slots in the FDT action routine vector with the procedure value of an upper-level FDT action routine

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.

If a driver does not explicitly invoke the FDT_INI macro, the FDT_BUF and FDT_ACT macros will automatically invoke it on the driver's behalf, resulting in an FDT named DRIVER$FDT.

The FDT_BUF macro builds the buffered function mask within an FDT from the specified list of I/O functions. The macro takes a single argument: a 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. (See Table 4–1 for a list of symbolic names for I/O functions defined by the $IODEF macro in SYS$LIBRARY:STARLET.MLB.)

If no buffered I/O functions are defined for the device, you can omit the FDT_BUF invocation.

In selecting the functions that are to be buffered, consider the following:

- Direct I/O is intended only for devices whose I/O operations always complete quickly. For example, although terminal I/O appears fast, users can prevent the I/O operation from completing by using Ctrl/S to halt the operation indefinitely; therefore, terminal I/O operations are buffered I/O.
- Use of direct I/O requires that the process pages containing the buffer be locked in memory. Locking pages in memory increases the overhead of swapping the process that contains the pages.
- Use of buffered I/O requires that the data be moved from the system buffer to the user buffer. Moving data requires additional time.
- Routines that manipulate data before delivering it to the user (for example, an interrupt service routine for a terminal) cannot gain access to the data if direct I/O is used. Therefore, transfers that require data manipulation must be buffered I/O.
- The operating system handles the quotas differently for direct I/O and buffered I/O, as described in the *OpenVMS System Manager's Manual*.
- Generally, direct-memory-access (DMA) devices use direct I/O, while programmed I/O devices use buffered I/O.

The FDT_ACT macro identifies the upper-level FDT action routine that processes one or more specified I/O function codes. An upper-level FDT action routine must either be an FDT completion routine or eventually transfer control to an FDT completion routine. An FDT completion routine either queues an I/O request packet (IRP) to a driver, inserts an IRP in the postprocessing queue, or aborts the I/O request. See Chapter 5 for additional information on upper-level FDT action routines, FDT support routines, and FDT completion routines.

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. Writers of OpenVMS VAX drivers should take special note of this difference between OpenVMS VAX and OpenVMS AXP FDT processing. On OpenVMS VAX systems, the $QIO system service processes entries in the order in which they appear in the FDT. When a function code is associated with more than one FDT routine, the system service sequentially calls every routine specified for the function code until a routine stops the scan by aborting, completing, or queuing an I/O request.

For a description of the FDT_INI, FDT_BUF, and FDT_ACT macros, and an example of their use, see the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

4.3.1 OpenVMS AXP I/O Function Codes

Table 4–1 lists the physical, logical, and virtual I/O function codes defined by the operating system. A complete list of function codes and values is contained in the macro $IODEF in SYS$LIBRARY:STARLET.MLB.

Table 4–1 I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_NOP†	No operation	–
IO$_UNLOAD	Unload drive (required by all disk drivers)	IO$_LOADMCODE† (load microcode), IO$_START_BUS† (start LAVc bus)

†Unsupported; subject to change without notice

(continued on next page)

Table 4–1 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_SEEK	Seek cylinder	IO$_SPACEFILE† (space files), IO$_STARTMPROC† (start microprocessor), IO$_STOP_BUS† (stop LAVc bus)
IO$_RECAL†	Recalibrate drive	IO$_DUPLEX† (enter duplex mode), IO$_STOP† (stop), IO$_DEF_COMP† (define network component)
IO$_DRVCLR†	Drive clear	IO$_INITIALIZE (initialize), IO$_MIMIC† (enter mimic mode), IO$_DEF_COMP_LIST† (define network component list)
IO$_RELEASE†	Release port	IO$_SETCLOCKP† (set clock—physical), IO$_START_ANALYSIS† (start LAVc failure analysis)
IO$_OFFSET†	Offset read heads	IO$_ERASETAPE† (erase tape), IO$_STARTDATAP† (start data transfer—physical), IO$_STOP_ANALYSIS† (stop LAVc failure analysis)
IO$_RETCENTER†	Return to center line	IO$_QSTOP† (queue stop request), IO$_START_MONITOR† (start LAVc channel monitor)
IO$_PACKACK	Pack acknowledgment (required by all disk drivers)	IO$_STOP_MONITOR† (stop LAVc channel monitor)
IO$_SEARCH	Search for sector	IO$_SPACERECORD† (space records), IO$_READRCT† (read replacement and caching table)
IO$_WRITECHECK	Write check data	–
IO$_WRITEPBLK	Write physical block	–
IO$_READPBLK	Read physical block	–
IO$_WRITEHEAD†	Write header and data	IO$_RDSTATS† (read statistics), IO$_CRESHAD† (create a shadow set)
IO$_READHEAD†	Read header and data	IO$_ADDSHAD† (add member to shadow set)
IO$_WRITETRACKD†	Write track data	IO$_COPYSHAD† (perform shadow set copy operations)
IO$_READTRACKD†	Read track data	IO$_REMSHAD† (remove member from shadow set)
IO$_AVAILABLE	Set device available (required by all disk drivers)	–
IO$_SETPRFPATH†	Set preferred path	–
IO$_DISPLAY†	Display MSCP/TMSCP volume label	–
IO$_DSE	Data security erase (and rewind)	–
IO$_REREADN†	Reread next	IO$_DISK_COPY_DATA† (disk copy data)
IO$_REREADP†	Reread previous	IO$_WHM† (write history management), IO$_AS_SETCHAR† (Asian set character)

†Unsupported; subject to change without notice

(continued on next page)

Table 4–1 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_WRITERET†	Write retry	IO$_WRITECHECKH† (write check header and data), IO$_AS_SENSE_CHAR† (Asian sense characteristics)
IO$_READPRESET†	Read in preset	IO$_STARTSPNDL† (start spindle)
IO$_SETCHAR	Set device characteristics	–
IO$_SENSECHAR	Sense device characteristics	–
IO$_WRITEMARK†	Write tape mark	IO$_COPYMEM† (copy memory), IO$_ PSXSETCHAR† (POSIX set characteristics)
IO$_WRTTMKR†	Write tape mark retry	IO$_DIAGNOSE† (diagnose), IO$_ SHADMV† (perform mount verification on shadow set), IO$_PSXSENSECHAR† (POSIX sense characteristics)
IO$_FORMAT	Format	IO$_CLEAN† (clean tape)
Logical I/O		
IO$_WRITELBLK	Write logical block	–
IO$_READLBLK	Read logical block	–
IO$_REWINDOFF	Rewind and set offline	IO$_READRCTL† (read RCT sector 0)
IO$_SETMODE	Set mode	–
IO$_REWIND	Rewind tape	–
IO$_SKIPFILE	Skip files	IO$_PSXSETMODE† (POSIX set mode)
IO$_SKIPRECORD	Skip records	IO$_PSXSENSEMODE† (POSIX sense mode)
IO$_SENSEMODE	Sense mode	–
IO$_WRITEOF	Write end of file	IO$_TTY_PORT_BUFIO† (Terminal port driver FDT routine for buffered I/O)
IO$_TTY_PORT†	Terminal port driver FDT routine	IO$_FREECAP† (return free capacity)
IO$_FLUSH†	Flush controller cache	IO$_AS_SETMODE† (Asian set mode)
IO$_READLCHUNK†	Read large logical block	IO$_AS_SENSEMODE† (Asian sense mode)
IO$_WRITELCHUNK†	Write large logical block	–
Virtual I/O		
IO$_WRITEVBLK	Write virtual block	–
IO$_READVBLK	Read virtual block	–
IO$_ACCESS	Access file	IO$_PSXWRITEVBLK† (POSIX write virtual block)
IO$_CREATE	Create file	–
IO$_DEACCESS	Deaccess file	IO$_PSXREADVBLK† (POSIX read virtual block)

†Unsupported; subject to change without notice

(continued on next page)

Table 4–1 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Virtual I/O		
IO$_DELETE	Delete file	–
IO$_MODIFY	Modify file	IO$_NETCONTROL† (X.25 network control function)
IO$_READPROMPT	Read terminal with prompt	IO$_SETCLOCK (set clock), IO$_AUDIO (CD-ROM audio)
IO$_ACPCONTROL	Miscellaneous ACP control	IO$_STARTDATA (start data)
IO$_MOUNT†	Mount volume	–
IO$_TTYREADALL†	Terminal read passall	–
IO$_TTYREADPALL†	Terminal read with prompt passall	–
IO$_CONINTREAD	Connect to interrupt read-only	–
IO$_CONINTWRITE	Connect to interrupt with write	–

†Unsupported; subject to change without notice

4.3.1.1 Defining Device-Specific Function Codes

You can also define device-specific function codes by equating the name of a device-specific function with the name of an existing function that is irrelevant to the device. The selected codes should, however, have a type (logical, physical, or virtual) that is appropriate for the function they represent. Also, user programs that issue $QIO requests specifying a device-specific code must similarly redefine the existing function. For example, the assembly code that follows defines three device-specific physical I/O function codes.

```
IO$_STARTCLOCK=IO$_ERASETAPE      ; Start interval clock
IO$_STOPCLOCK=IO$_OFFSET          ; Stop interval clock
IO$_STARTDATA=IO$_SPACEFILE       ; Start data acquisition
```

5
Writing FDT Routines

A driver performs device-specific I/O function preprocessing to validate arguments specified in the original call to the $QIO system service, to complete certain types of function processing such as set mode and sense mode operations, and to prepare to service functions involving a device transaction. Driver I/O function preprocessing on OpenVMS AXP systems often requires the cooperative efforts of upper-level FDT action routines, FDT support routines, and FDT completion routines.

An **upper-level FDT action routine** is a routine listed in a driver's function decision table (FDT) as a result of the driver's invocation of the FDT_ACT macro (see Chapter 4 for more information). FDT dispatching code in the $QIO system service calls an upper-level FDT action routine, passing to it the addresses of the I/O request packet (IRP), process control block (PCB), unit control block (UCB), and channel control block (CCB). An upper-level FDT action routine must return SS$_FDT_COMPL status to the $QIO system service. (See the *OpenVMS AXP Step 2 Device Driver Reference Manual* for a full description of the formal interface to an upper-level FDT action routine.)

The $QIO system service uses the FDT to determine which upper-level FDT action routine to call to initiate driver preprocessing of a specific I/O request. Often a driver can use one of the system-provided upper-level FDT routines, as described in Section 5.2.1. This practice encourages the use of well debugged routines and minimizes driver size. However, if the I/O function requires preprocessing that is unique to the device the driver controls, the driver includes an upper-level FDT routine that services the function in a device-dependent manner.

An upper-level FDT action routine sometimes calls an intermediate FDT routine, known as an **FDT support routine**. An FDT support routine performs some discrete action on behalf of an upper-level action routine, such as determining whether a user buffer is accessible, locking a user buffer in memory, and reformatting data into buffers in the system address space. Often, an FDT support routine calls an FDT completion routine.

There is no formal interface for calling FDT support routines, except the FDT support routines provided by OpenVMS. These routines are listed in Table 2–3 and described in the *OpenVMS AXP Step 2 Device Driver Reference Manual*. Rather, a driver-specific FDT support routine and its caller agree on the mechanism used to accomplish the call (either a standard call or JSB) and the participating data and registers. The $QIO system service never calls an FDT support routine directly to process a given I/O function.

To conclude the preprocessing of an I/O function, a driver FDT routine (either the upper-level FDT action routine or an FDT support routine) calls a system-provided **FDT completion routine**. An FDT completion routine places the status return to the $QIO system service in the FDT context (FDT_CONTEXT) structure and returns SS$_FDT_COMPL status to its caller. The caller must

process the returned status. Eventually, the driver's upper-level FDT routine exits FDT processing by returning control to the $QIO system service.

Note that FDT support routines and FDT completion routines return status to their callers. Each FDT routine that participates in the processing of an I/O function should examine the status value returned to it by any routine it calls and should reflect this status to the routine that called it.

5.1 Context of Driver FDT Processing

The $QIO system service executes in the context of the process that issues the I/O request, but in kernel mode and at IPL$_ASTDEL. The process is executing in kernel mode because the dispatching of the $QIO system service executes a CHMK PALcode instruction. Process context allows the $QIO system service and driver FDT routines to access process address space. Because the $QIO system service expects FDT routines to preserve this context, an FDT routine observes the following conventions:

- An FDT routine must not call system services or OpenVMS RMS routines.
- It must not lower IPL below IPL$_ASTDEL. If an FDT routine raises IPL, it must obtain any appropriate spin lock, and it must lower IPL to IPL$_ ASTDEL before exiting, releasing any acquired spin lock.
- It should not access device registers because the device might be active.
- It should not alter the stack without restoring its original state before returning to its caller.
- It must preserve the contents of all nonscratch registers defined by the *OpenVMS Calling Standard* for AXP systems.
- It should exercise restraint when modifying the UCB. Routines usually access the UCB while holding the associated fork lock at driver fork IPL to synchronize modifications, and FDT routines do not execute with such synchronization. Drivers containing FDT routines that access device registers or carelessly modify the UCB risk unpredictable operation or a system failure.

5.2 Upper-Level FDT Action Routines

A Step 2 OpenVMS AXP device driver provides a single upper-level FDT routine for each I/O function code it processes. The $QIO system service uses the low-order six bits of the I/O function code as an index into the FDT action routine vector. As described in Chapter 4, any vector slots corresponding to a driver-supported function contain the procedure value of either a system or driver-provided upper-level FDT action routine. Those which correspond to unsupported functions contain the procedure value of the system upper-level FDT action routine EXE$ILLIOFUNC.

The $QIO system service transfers control to an upper-level FDT action routine using the following interface:

```
status = driver_FDT_routine(irp,pcb,ucb,ccb)
```

The parameters include the addresses of:

- the I/O request packet (IRP) for the current I/O request
- the process control block (PCB) of the current process

- the unit control block (UCB) of the device assigned to the process-I/O channel specified as an argument to the $QIO request
- the channel control block (CCB) that describes the process-I/O channel

An upper-level FDT action routine must return SS$_FDT_COMPL status to its caller, the FDT dispatching code in the $QIO system service.

Use the $DRIVER_FDT_ENTRY macro to declare the call entry point of an upper-level FDT action routine.

The $DRIVER_FDT_ENTRY macro specifies, by default, that registers R2 through R15 be preserved. For certain FDT entry points, you can specify a much smaller set of registers — **preserve=<R2,R9,R10,R11>** is usually sufficient. (These registers are allowed to be scratched by Step 1 FDT routines.)

You can safely follow this recommendation only if the FDT processing initiated by the upper-level FDT action routine avoids the situation in which a call initiated by a JSB instruction is concluded by a RET instruction instead of an RSB (a situation for which the MACRO-32 compiler generates a warning). A "RET under JSB" can occur in FDT processing if the upper-level FDT routine issues a JSB to an FDT support routine that invokes an FDT completion macro (see Table 2–2) without specifying **do_ret=NO**. The additional RET instruction generated by a default invocation of the macro would return control back to FDT dispatching code in the $QIO system service, and risks the destruction of register context required by that code.

By default, the $DRIVER_FDT_ENTRY macro also loads the IRP address into R3, PCB address into R4, UCB address into R5, and CCB address into R6 to approximate the register entry values for step 1 FDT routines. This register loading can be disabled by an optional parameter.

Certain parameters that are passed in registers to an OpenVMS VAX FDT routine are not provided to an OpenVMS AXP upper-level FDT action routine. These include the following:

- The address of the first function-dependent argument (**p1**) specified in I/O request, passed to an OpenVMS VAX FDT routine at AP.

  An OpenVMS AXP driver obtains the contents of the function-dependent arguments from IRP$L_QIO_P*n*, where *n* is a parameter number from 1 through 6.

- The bit number of the user-specified I/O function code, passed to an OpenVMS VAX FDT routine in R7

  An OpenVMS AXP driver obtains the user-supplied function code from six low-order bits of IRP$L_FUNC.

- The address of the current entry in the FDT, passed to an OpenVMS VAX FDT routine in R8

  An OpenVMS AXP driver obtains the address of the *start* of the FDT (as depicted in Figure 4–1) from DDT$PS_FDT2. It obtains the address of the DDT from UCB$L_DDT.

  Note that the format of the Step 2 FDT differs drastically from the format of the Step 1 FDT. Refer to the $FDTDEF macro in SYS$LIBRARY:LIB.MLB if your driver needs specific information about the structure of the FDT.

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*(AP), where *xxx* is the three-letter structure acronym, to access the input parameters.

Before exiting, the upper-level FDT action routine takes steps to complete FDT processing. Typically, these steps include:

- Calling an FDT completion routine, which takes steps to complete the processing of an I/O request or to deliver the I/O request to the driver. An FDT completion routine typically provides the final $QIO completion status in the FDT_CONTEXT structure and returns SS$_FDT_COMPL warning status to its caller. SS$_FDT_COMPL status is a warning that FDT processing has been completed and that the IRP is no longer accessible to FDT processing. (For instance, the IRP may have been deallocated or queued to the driver's start-I/O routine, which accesses the IRP at fork IPL.)
- Returning SS$_FDT_COMPL status to its caller, FDT dispatching code in the $QIO system service.

Section 5.2.2 describes the FDT completion routines provided by OpenVMS.

If the preprocessing of a given I/O function requires the execution of multiple upper-level FDT action routines, a Step 2 driver should provide a *composite* FDT function which sequentially calls each of the required FDT routines as long as the returned status is successful.

5.2.1 System-Provided Upper-Level FDT Routines

The system-provided upper-level FDT routines perform I/O request validation that is common to many devices. Whenever possible, drivers should take advantage of these routines. All of the system FDT routines listed in Table 5–1 transfer control to EXE_STD$QIODRVPKT, EXE_STD$FINISHIO, or EXE_STD$ABORTIO to complete the I/O request. These FDT completion routines, as described in Section 5.2.2, place final $QIO completion status in the FDT_CONTEXT structure, and return SS$_FDT_COMPL status to the upper-level FDT action routine. All upper-level FDT action routines return to the FDT dispatching code in the $QIO system service.

For additional information about system-provided FDT routines, see the specific routine descriptions in the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

Table 5–1 System-Provided Upper-Level FDT Action Routines

FDT Routine	Function	Completion Routine Used
ACP_STD$ACCESS	Accesses and creates ACP function processing	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$ACCESSNET	Connects to network function processing	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO

(continued on next page)

Table 5–1 (Cont.) System-Provided Upper-Level FDT Action Routines

FDT Routine	Function	Completion Routine Used
ACP_STD$DEACCESS	Deaccesses ACP function processing	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$MODIFY	Deletes and modifies ACP function processing	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$MOUNT	Initiates ACP mount function processing	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$READBLK	Processes a read block ACP function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$WRITEBLK	Processes a write block ACP function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
EXE$ILLIOFUNC	Processes I/O functions not supported by the driver	Calls EXE_STD$ABORTIO
EXE_ STD$LCLDSKVALID	Processes an IO$_PACKACK, IO$_ AVAILABLE, or IO$_UNLOAD function on a local disk	EXE_STD$FINISHIO or EXE_ STD$QIODRVPKT
EXE_STD$MODIFY	Processes a logical-read/write or physical-read/write function for a read and write direct I/O operation to a user-specified buffer	Calls EXE_STD$ABORTIO if an error occurs (for instance, if the user I/O buffers cannot be accessed or cannot be locked in memory); otherwise, calls EXE_ STD$QIODRVPKT
EXE_STD$ONEPARM	Processes a nontransfer I/O function code that has one parameter associated with it	Calls EXE_STD$QIODRVPKT
EXE_STD$READ	Processes a logical-read or physical-read function for a direct I/O operation	Calls EXE_STD$ABORTIO if an error occurs (for instance, if the user I/O buffers cannot be accessed or locked in memory); otherwise, calls EXE_ STD$QIODRVPKT
EXE_STD$SENSEMODE	Processes the sense-device-mode and sense-device-characteristics functions by reading fields of the UCB	Calls EXE_STD$FINISHIO
EXE_STD$SETCHAR[1]	Processes the set-device-mode and set-device-characteristics functions	Calls EXE_STD$FINISHIO
EXE_STD$SETMODE[1]	Processes the set-device-mode and set-device-characteristics functions by creating a driver fork process	Calls EXE_STD$ABORTIO if an error occurs; otherwise, calls EXE_STD$QIODRVPKT
EXE_STD$WRITE	Processes a logical-write or physical-write function for a direct I/O operation	Calls EXE_STD$ABORTIO if an error occurs (for instance, if the user I/O buffers cannot be accessed or locked in memory); otherwise, calls EXE_ STD$QIODRVPKT

[1]If setting device characteristics requires no device activity or requires no synchronization with fork processing, the driver's FDT entry can specify EXE_STD$SETCHAR; otherwise, it must specify EXE_STD$SETMODE.

(continued on next page)

Table 5–1 (Cont.) System-Provided Upper-Level FDT Action Routines

FDT Routine	Function	Completion Routine Used
EXE_STD$ZEROPARM	Processes a nontransfer I/O function code that has no associated parameters	Calls EXE_STD$QIODRVPKT

Driver FDT processing selects an I/O completion path based on the following factors:

- Whether it needs to call another FDT routine to perform additional function-specific processing.
- Whether an error is found in the I/O request.
- Whether the operation is complete.
- Whether the I/O operation requires and is ready for device activity.

Any specific I/O function can be processed by only one upper-level FDT action routine. Although an upper-level routine can call any number of subsequent routines, it must eventually complete I/O processing by returning SS$_FDT_COMPL status in R0 (and final $QIO completion status in the FDT_CONTEXT structure) to its caller, FDT dispatching code in the $QIO system service.

The system-provided upper-level FDT routines, as discussed in Table 5–1, all call an FDT completion routine that queues an IRP, completes an I/O request, or aborts an I/O request. These FDT completion routines insert the final $QIO system service status in the FDT_CONTEXT structure and return SS$_FDT_COMPL warning status to the upper-level FDT action routine. The upper-level FDT action routine returns these status values to its caller, FDT dispatching code in the $QIO system service.

5.2.2 FDT Exit Paths

An upper-level FDT action routine completes I/O function invoking one of the completion macros listed in Table 5–2.

Each of these macros expects the same input registers as the corresponding Step 1 FDT exit routine. These macros expand to code that calls the appropriate step 2 FDT completion routine, passing the implicit input registers in the correct call interface parameter order. These FDT completion macros, by default, generate a RET instruction in their expansion that causes control to be returned to FDT dispatching code in the $QIO system service.

Thus, for example, a Step 2 driver can use:

```
$CALL_ABORTIO
```

as a replacement for:

```
JMP G^EXE$ABORTIO
```

Table 5–2 FDT Completion Macros and Associated Routines

Macro	Operation
$ABORTIO	Calls EXE_STD$ABORTIO to abort an I/O request. An FDT routine that discovers a device-independent error should always use this method of exiting. Inability to gain access to a data buffer or an error in the specification of the I/O request are examples of device-independent errors.
$ALTQUEPKT	Calls EXE_STD$ALTQUEPKT to call an alternate start-I/O routine in the driver (specified in the driver dispatch table at offset DDT$L_ALTSTART) that synchronizes requests for activity on a device unit and initiates the processing of I/O requests. The FDT routine uses this exit method when it has successfully completed all driver preprocessing and the request requires device activity. However, in contrast to EXE_STD$QIODRVPKT, EXE_STD$ALTQUEPKT bypasses the device unit's pending-I/O queue and the device busy flag; thus, the driver is activated regardless of whether the device unit is busy. A driver that can handle two or more I/O requests simultaneously uses this exit method. Be aware that programming a device driver to process simultaneous I/O requests requires detailed knowledge of the internal design of the operating system. A driver that uses the $ALTQUEPKT macro must not only maintain its internal queues but must also synchronize those queues with the unit's pending-I/O queue, which the operating system maintains. In addition, if a driver processes more than one IRP at the same time, it must use separate fork blocks. Such a driver completes the processing of I/O requests by using the $POST macro (which calls COM_STD$POST). This routine places each IRP in the systemwide I/O postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. For more information about COM_STD$POST, see the *OpenVMS AXP Step 2 Device Driver Reference Manual*. When the alternate start-I/O routine finishes, it returns control to EXE_STD$ALTQUEPKT by issuing a RET instruction. EXE_STD$ALTQUEPKT then returns to the FDT routine that called it. The FDT routine performs any necessary postprocessing, including placing SS$_NORMAL status in the FDT_CONTEXT structure and returning (with the RET instruction) to FDT dispatching code in the $QIO system service
$FINISHIO	Moves the contents of R0 and R1 to IRP$L_IOST1 and IRP$L_IOST2, respectively and calls EXE_STD$FINISHIO to insert the IRP in the I/O postprocessing queue. EXE_STD$DINISHIO returns SS$_FDT_COMPL status to the $QIO system service and SS$_NORMAL status (in the FDT_CONTEXT structure) to the caller of $QIO. An FDT routine that discovers a device-dependent error should always return status using $FINISHIO or $FINISHIOC.
$FINISHIOC	Calls EXE_STD$FINISHIO to perform the same operations as $FINISHIO except $FINISHIOC clears the second longword of the final I/O status.
$FINISHIO_NOIOST	Calls EXE_STD$FINISHIO to perform the same operations as $FINISHIO except $FINISHIO_NOIOST does not fill in the I/O status fields of the IRP.

(continued on next page)

Table 5–2 (Cont.) FDT Completion Macros and Associated Routines

Macro	Operation
$IORSNWAIT	Calls EXE_STD$IORSNWAIT. Reserved to Digital.
$QIOACPPKT	Calls EXE_STD$QIOACPPKT. Reserved to Digital.
$QIODRVPKT	Calls EXE_STD$QIODRVPKT to transfer control to a system routine (EXE_STD$INSIOQ) that either delivers an IRP immediately to a driver's start-I/O routine or places the IRP in a pending-I/O queue waiting for driver servicing. The FDT routine uses this FDT completion routine if all preprocessing is complete, if no errors are found in the specification of an I/O request, and if device activity, synchronized access to the device's UCB, or synchronized access to device registers is required to complete the I/O request. Common examples of such a request are read and write functions. EXE_STD$INSIOQ transfers control to the device driver's start-I/O routine only if the device unit is currently idle. If the device unit is busy, EXE_STD$INSIOQ inserts the IRP in a priority-ordered queue of IRPs waiting for the unit. Once an FDT routine transfers control to EXE_STD$QIODRVPKT, no driver code that further processes the I/O request can refer to process virtual address space. When a device driver's start-I/O routine gains control, the process that queued the I/O request might no longer be the mapped process. Therefore, the driver must assume that all information regarding the I/O request is in the UCB or the IRP and that all buffer addresses in the UCB are either system addresses or page-frame numbers that can be interpreted in any process context. For direct I/O operations, FDT routines also must have locked all user buffer pages in physical memory because paging cannot occur at driver fork level or higher interrupt priority levels. The process virtual address space is not guaranteed to be mapped again until the operating system delivers a special kernel-mode asynchronous system trap (AST) to the requesting process as part of I/O postprocessing.

5.3 FDT Routines for System Direct I/O

The operating system provides two standard FDT routines that are applicable for direct I/O operations: EXE_STD$READ and EXE_STD$WRITE. When called by the driver, these routines completely prepare a direct I/O read or write request. Thus, a driver that uses these routines eliminates the need for its own device-specific FDT routines.

EXE_STD$READ and EXE_STD$WRITE are described in the *OpenVMS AXP Step 2 Device Driver Reference Manual*.

5.4 FDT Routines for System Buffered I/O

Device drivers for buffered I/O operations generally contain their own device-specific FDT routines.

An FDT routine for a buffered I/O data transfer operation should confirm either read or write access to the user's buffer and allocate a buffer in system space. Sections 5.4.1 and 5.4.2 describe these tasks.

An FDT routine for a buffered I/O operation that does not involve data transfer accesses the function-dependent parameters of the $QIO request (**p1** to **p6**) from IRP$L_QIO_P*n*, where *n* is a parameter number from one to six. It performs any necessary preprocessing and uses one of the exit methods listed in Section 5.2.2.

5.4.1 Checking Accessibility of the User's Buffer

First the FDT routine invokes the $READCHK or $WRITECHK macros (which call EXE_STD$READCHK or EXE_STD$WRITECHK, respectively) to confirm write or read access to the user's buffer. Both of these routines write the transfer byte count into IRP$L_BCNT. EXE_STD$READCHK also sets IRP$V_FUNC in IRP$L_STS to indicate that it is a read function.

5.4.2 Allocating the System Buffer

Next, the FDT routine allocates a system buffer in the following manner:

1. It adds 12 bytes to the byte count passed in the **p2** argument of the user's I/O request (obtained from IRP$L_QIO_P2), thus accommodating the standard size of a system buffer header. This is the total system buffer size.
2. It issues a JSB to EXE$DEBIT_BYTCNT_ALO to ensure that the job has sufficient remaining byte count quota to allow its use of the requested buffer. If the job has sufficient quota, EXE$DEBIT_BYTCNT_ALO allocates the requested buffer from nonpaged pool, writes the buffer's size and type into its third longword, and subtracts the system buffer size from JIB$L_BYTCNT.

The operating system also supplies the routines EXE$DEBIT_BYTCNT_BYTLM_ALO, EXE$DEBIT_BYTCNT(_NW), EXE$DEBIT_BYTCNT_BYTLM(_NW), and EXE_STD$ALLOCBUF, which perform the same type of work as EXE$DEBIT_BYTCNT_ALO. These routines are fully described in the *OpenVMS VAX Device Support Reference Manual*.

Once the buffer is allocated, the FDT routine takes the following steps:

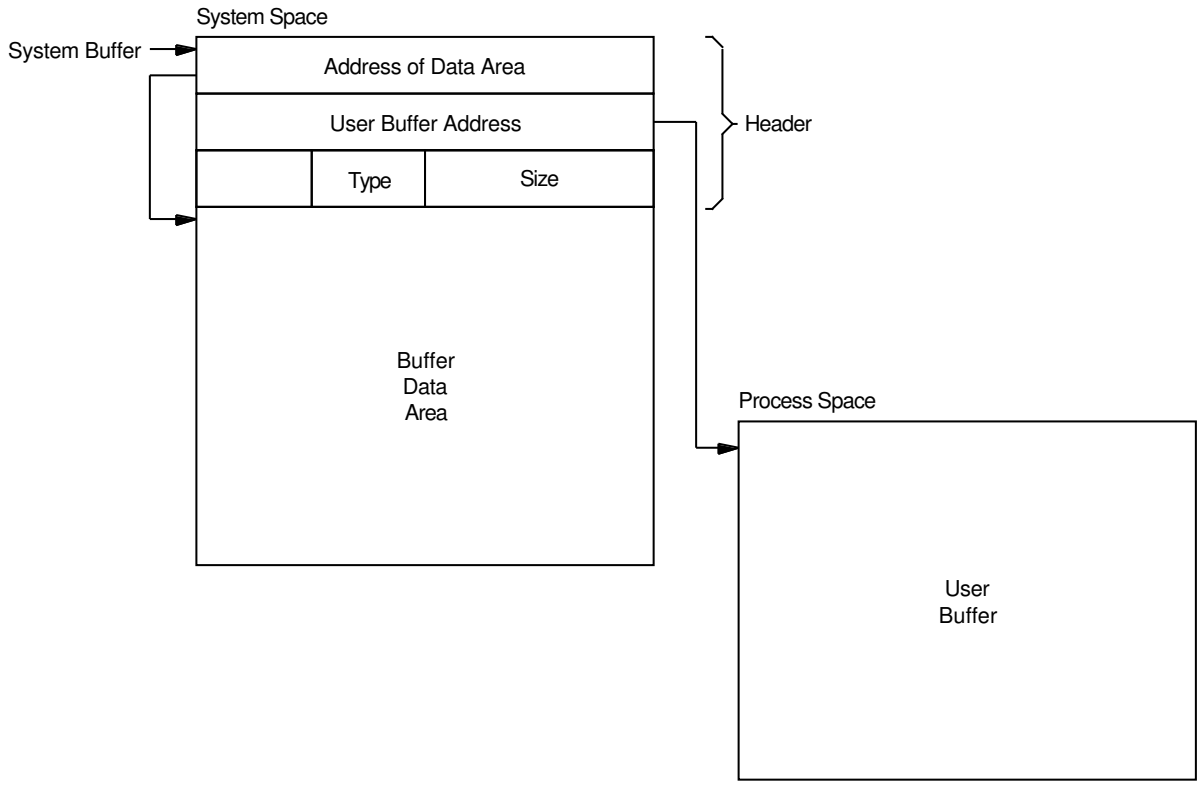
1. Loads the address of the system buffer into IRP$L_SVAPTE.
2. Loads the total size of the system buffer into IRP$L_BOFF.
3. Stores the starting address of the system buffer data area in the first longword of the buffer header.
4. Stores the user's buffer address in the second longword of the header.
5. Copies data from the user buffer to the system buffer if the I/O request is a write operation.

At this point, the buffers are ready for the transfer. Figure 5–1 illustrates the format of the system buffer.

5.4.3 Buffered-I/O Postprocessing

When the transfer finishes, the driver returns control to the operating system for completion of the I/O request. The driver writes the final request status in the low-order word of R0. Use of the high-order word of R0 and the longword of R1 is driver specific. Certain drivers use these fields to report a transfer byte count, for example.

Figure 5–1 Format of System Buffer for a Buffered-I/O Read Function



ZK–0927–GE

The driver must leave the buffer header intact; I/O postprocessing relies on the header's accuracy. When system I/O postprocessing gains control, it performs three steps:

1. Issues a JSB instruction to EXE$CREDIT_BYTCNT to add the value in IRP$L_BOFF to JIB$L_BYTCNT, thus updating the user's byte count quota.
2. If IRP$L_SVAPTE is nonzero, assumes a system buffer was allocated and checks to see whether IRP$V_FUNC is set in IRP$L_STS.
3. If IRP$V_FUNC is clear, deallocates the system buffer used for the write operation; if IRP$V_FUNC is set, the special kernel-mode AST copies the data to the user's buffer and then deallocates the buffer in addition to performing other kernel-mode AST functions.

The special kernel-mode AST performs the following steps to complete a buffered read operation:

1. Obtains the address of the system buffer from IRP$L_SVAPTE.
2. Obtains the number of bytes to write to the user's buffer from IRP$L_BCNT.
3. Obtains the address of the user's buffer from the second longword of the system buffer header.
4. Checks for write accessibility on all pages of the user's buffer.
5. Copies the data from the system buffer to the process buffer.

6. Deallocates the system buffer. Note that the system uses the size listed in the buffer's header to deallocate the buffer.

Index

A

ACP$ACCESS routine, 2–3, 2–15
ACP$ACCESSNET routine, 2–3, 2–15
ACP$DEACCESS routine, 2–3, 2–15
ACP$MODIFY routine, 2–3, 2–15
ACP$MOUNT routine, 2–3, 2–15
ACP$READBLK routine, 2–3, 2–15
ACP$WRITEBLK routine, 2–3, 2–15
ACP_STD$ACCESS routine, 2–3, 2–10, 2–15
ACP_STD$ACCESSNET routine, 2–3, 2–10, 2–15
ACP_STD$DEACCESS routine, 2–3, 2–10, 2–15
ACP_STD$MODIFY routine, 2–3, 2–10, 2–15
ACP_STD$MOUNT routine, 2–3, 2–10, 2–15
ACP_STD$READBLK routine, 2–3, 2–10, 2–15
ACP_STD$WRITEBLK routine, 2–3, 2–10, 2–15
Alternate start I/O routine
  address, 4–4
AST (asynchronous system trap)
  special kernel-mode, 5–10 to 5–11

B

BLISS drivers
  converting to Step 2, 1–3
Buffer
  allocating, 5–9
  data area, 5–9
  deallocating, 5–10
  format, 5–9
  header area, 5–9
  locking, 4–5
  size, 5–9
  storing address of, 5–9
  testing accessibility of, 5–9
Buffered function mask, 4–4, 4–5
Buffered I/O
  FDT routines for, 5–8 to 5–11
  functions, 4–4
  postprocessing, 5–9 to 5–11
  reasons for using, 4–5
Buffered I/O function
  defining, 2–2

C

C drivers
  writing Step 2, 1–3
Call-based system routine
  interface, 1–2
  naming, 1–2
CALL_ABORTIO macro, 2–5, 2–10, 2–15
CALL_ALLOCBUF macro, 2–10
CALL_ALLOCEMB macro, 2–10
CALL_ALLOCIRP macro, 2–10
CALL_ALTQUEPKT macro, 2–5, 2–11
CALL_ALTREQCOM macro, 2–12
CALL_BROADCAST macro, 2–12
CALL_CANCELIO macro, 2–12
CALL_CARRIAGE macro, 2–11
CALL_CHKCREACCES macro, 2–11
CALL_CHKDELACCES macro, 2–11
CALL_CHKEXEACCES macro, 2–11
CALL_CHKLOGACCES macro, 2–11
CALL_CHKPHYACCES macro, 2–11
CALL_CHKRDACCES macro, 2–11
CALL_CHKWRTACCES macro, 2–11
CALL_CLONE_UCB macro, 2–12, 2–16
CALL_COPY_UCB macro, 2–12
CALL_CREDIT_UCB macro, 2–12
CALL_CVTLOGPHY macro, 2–13
CALL_CVT_DEVNAM macro, 2–13
CALL_DELATTNAST macro, 2–10
CALL_DELATTNASTP macro, 2–10
CALL_DELCTRLAST macro, 2–10
CALL_DELCTRLASTP macro, 2–10
CALL_DELETE_UCB macro, 2–13
CALL_DEVICEATTN macro, 2–10
CALL_DEVICERR macro, 2–10
CALL_DEVICTMO macro, 2–10
CALL_DIAGBUFILL macro, 2–13
CALL_DRVDEALMEM macro, 2–10
CALL_FILSPT macro, 2–13
CALL_FINISHIO macro, 2–5, 2–11, 2–15
CALL_FINISHIOC macro, 2–5, 2–15
CALL_FINISHIO_NOIOST macro, 2–5
CALL_FLUSHATTNS macro, 2–10
CALL_FLUSHCTRLS macro, 2–10

CALL_GETBYTE macro, 2–13
CALL_INITBUFWIND macro, 2–13
CALL_INITIATE macro, 2–13
CALL_INSERT_IRP macro, 2–11
CALL_INSIOQ macro, 2–11
CALL_INSIOQC macro, 2–11
CALL_IOLOCK macro, 2–14
CALL_IOLOCKR macro, 2–14
CALL_IOLOCKW macro, 2–14
CALL_IORSNWAIT macro, 2–5, 2–11, 2–15
CALL_LINK_UCB macro, 2–13, 2–16
CALL_MAPVBLK macro, 2–13
CALL_MNTVER macro, 2–13
CALL_MNTVERSIO macro, 2–11
CALL_MODIFYLOCK macro, 2–6, 2–11, 2–15
CALL_MODIFYLOCK_ERR macro, 2–6, 2–11, 2–15
CALL_MOUNT_VER macro, 2–11
CALL_MOVFRUSER macro, 2–13
CALL_MOVFRUSER2 macro, 2–13
CALL_MOVTOUSER macro, 2–13
CALL_MOVTOUSER2 macro, 2–13
CALL_PARSDEVNAM macro, 2–13
CALL_POST macro, 2–10
CALL_POST_IRP macro, 2–13
CALL_POST_NOCNT macro, 2–10
CALL_PTETOPFN macro, 2–13
CALL_QIOACPPKT macro, 2–5, 2–12, 2–15
CALL_QIODRVPKT macro, 2–5, 2–12, 2–15
CALL_QNXTSEG1 macro, 2–13
CALL_QXQPPKT macro, 2–12
CALL_READCHK macro, 2–6, 2–12, 2–15
CALL_READCHKR macro, 2–6, 2–12, 2–15
CALL_READLOCK macro, 2–6, 2–12, 2–15
CALL_READLOCK_ERR macro, 2–6, 2–12
CALL_RELEASEMB macro, 2–10
CALL_SEARCHDEV macro, 2–13
CALL_SEARCHINT macro, 2–13
CALL_SETATTNAST macro, 2–6, 2–15
CALL_SETCTRLAST macro, 2–6, 2–10, 2–15
CALL_SEVER_UCB macro, 2–13
CALL_SIMREQCOM macro, 2–13
CALL_SNDEVMSG macro, 2–12
CALL_SSETATTNAST macro, 2–10
CALL_THREADCRB macro, 2–13
CALL_UNLOCK macro, 2–14
CALL_WRITECHK macro, 2–6, 2–12, 2–16
CALL_WRITECHKR macro, 2–6, 2–12, 2–16
CALL_WRITELOCK macro, 2–6, 2–12, 2–16
CALL_WRITELOCK_ERR macro, 2–6, 2–12, 2–16
CALL_WRTMAILBOX macro, 2–12
Cancel I/O routine
  address, 4–3
Cloned UCB routine
  address, 4–4
COM$SETATTNAST routine, 2–6, 2–15
COM$SETCTRLAST routine, 2–6, 2–15
COM_STD$DELATTNAST routine, 2–10
COM_STD$DELATTNASTP routine, 2–10
COM_STD$DELCTRLAST routine, 2–10
COM_STD$DELCTRLASTP routine, 2–10
COM_STD$DRVDEALMEM routine, 2–10
COM_STD$FLUSHATTNS routine, 2–10
COM_STD$FLUSHCTRLS routine, 2–10
COM_STD$POST routine, 2–10
COM_STD$POST_NOCNT routine, 2–10
COM_STD$SETATTNAST routine, 2–6, 2–10, 2–15
COM_STD$SETCTRLAST routine, 2–6, 2–10, 2–15
Controller initialization routine
  address, 4–3

D

Data structure
  initializing, 4–1
DDT (driver dispatch table)
  creating, 4–3 to 4–4
DDTAB macro, 2–1
Device characteristics, 5–5
Device driver
  entry points, 4–3 to 4–4
  loading, 4–1
  maximum number of supported units, 4–2
  program sections, 4–3
  Step 1, 1–1
  Step 2, 1–1
Device mode, 5–5
Diagnostic buffer
  specifying, 4–4
Direct I/O
  FDT routines for, 5–5, 5–8
  reasons for using, 4–5
Disk driver, 5–4
DMA transfer
  using direct I/O in, 4–5
DPT (driver prologue table)
  creating, 4–1 to 4–3
DPTAB macro, 2–1, 4–1
DPT_STORE macro, 4–3
$DRIVER_ALTSTART_ENTRY macro, 2–1
$DRIVER_CANCEL_ENTRY macro, 2–1
$DRIVER_CANCEL_SELECTIVE_ENTRY macro, 2–1
$DRIVER_CHANNEL_ASSIGN_ENTRY macro, 2–1
$DRIVER_CLONEDUCB_ENTRY macro, 2–1
$DRIVER_CTRLINIT_ENTRY macro, 2–1
$DRIVER_DELIVER_ENTRY macro, 2–1
$DRIVER_ERRRTN_ENTRY macro, 2–4

$DRIVER_FDT_ENTRY macro, 2–1, 2–4, 3–2, 5–3
$DRIVER_MNTVER_ENTRY macro, 2–1
$DRIVER_REGDUMP_ENTRY macro, 2–1
$DRIVER_START_ENTRY macro, 2–1
$DRIVER_UNITINIT_ENTRY macro, 2–1

E

Entry points, 1–1
  defining, 2–1
  returning from, 2–2
ERL_STD$ALLOCEMB routine, 2–10
ERL_STD$DEVICEATTN routine, 2–10
ERL_STD$DEVICERR routine, 2–10
ERL_STD$DEVICTMO routine, 2–10
ERL_STD$RELEASEMB routine, 2–10
Error message buffer
  specifying size, 4–4
EXE$ABORTIO routine, 2–5, 2–15
EXE$ALTQUEPKT routine, 2–5
EXE$CREDIT_BYTCNT, 5–10
EXE$DEBIT_BYTCNT_ALO, 5–9
EXE$DEBIT_BYTCNT_BYTLM, 5–9
EXE$DEBIT_BYTCNT_BYTLM_ALO, 5–9
EXE$FINISHIO routine, 2–5, 2–15
EXE$FINISHIOC routine, 2–5, 2–15
EXE$ILLIOFUNC routine, 2–3, 4–4
EXE$IORSNWAIT routine, 2–5, 2–15
EXE$KP_STARTIO routine, 1–2, 2–1, 2–15
EXE$LCLDSKVALID routine, 2–3, 2–15
EXE$MODIFY routine, 2–3, 2–15
EXE$MODIFYLOCK routine, 2–6, 2–15
EXE$MODIFYLOCK_ERR, 2–6
EXE$MODIFYLOCK_ERR routine, 2–15
EXE$ONEPARM routine, 2–3, 2–15
EXE$QIOACPPKT routine, 2–5, 2–15
EXE$QIODRVPKT routine, 2–5, 2–15
EXE$QIORETURN routine, 2–15
EXE$READ routine, 2–3, 2–15
EXE$READCHK, 5–9
EXE$READCHK routine, 2–6, 2–15
EXE$READCHKR routine, 2–6, 2–15
EXE$READLOCK routine, 2–6, 2–15
EXE$READLOCK_ERR routine, 2–6, 2–15
EXE$SENSEMODE routine, 2–3, 2–16
EXE$SETCHAR routine, 2–3, 2–16
EXE$SETMODE routine, 2–3, 2–16
EXE$WRITE routine, 2–3, 2–16
EXE$WRITECHK, 5–9
EXE$WRITECHK routine, 2–6, 2–16
EXE$WRITECHKR routine, 2–6, 2–16
EXE$WRITELOCK routine, 2–6, 2–16
EXE$WRITELOCK_ERR routine, 2–6, 2–16
EXE$ZEROPARM routine, 2–3, 2–16
EXE_STD$ABORTIO routine, 2–5, 2–10, 2–15, 4–4
EXE_STD$ALLOCBUF, 5–9
EXE_STD$ALLOCBUF routine, 2–10
EXE_STD$ALLOCCEB routine, 2–10
EXE_STD$ALLOCIRP routine, 2–10
EXE_STD$ALLOCJIB routine, 2–10
EXE_STD$ALLOCPCB routine, 2–10
EXE_STD$ALLOCTQE routine, 2–10
EXE_STD$ALONONPAGED routine, 2–11
EXE_STD$ALOPHYCNTG routine, 2–11
EXE_STD$ALTQUEPKT routine, 2–5, 2–11
EXE_STD$CARRIAGE routine, 2–11
EXE_STD$CHKCREACCES routine, 2–11
EXE_STD$CHKDELACCES routine, 2–11
EXE_STD$CHKEXEACCES routine, 2–11
EXE_STD$CHKLOGACCES routine, 2–11
EXE_STD$CHKPHYACCES routine, 2–11
EXE_STD$CHKPRO_INT routine, 2–11
EXE_STD$CHKRDACCES routine, 2–11
EXE_STD$CHKWRTACCES routine, 2–11
EXE_STD$CREDIT_BYTCNT_BYTLM routine, 2–11
EXE_STD$CVT_IPID_TO_EPID routine, 2–11
EXE_STD$CVT_IPID_TO_PCB routine, 2–11
EXE_STD$DEANONPAGED routine, 2–11
EXE_STD$DEANONPGDSIZ routine, 2–11
EXE_STD$DEBIT_BYTCNT routine, 2–11
EXE_STD$DEBIT_BYTCNT_ALO routine, 2–11
EXE_STD$DEBIT_BYTCNT_BYTLM_NW routine, 2–11
EXE_STD$FINISHIO, 5–5
EXE_STD$FINISHIO routine, 2–5, 2–11, 2–15
EXE_STD$FINISHIOC routine, 2–11
EXE_STD$INSERT_IRP routine, 2–11
EXE_STD$INSIOQ routine, 2–11
EXE_STD$INSIOQC routine, 2–11
EXE_STD$INSTIMQ routine, 2–11
EXE_STD$IORSNWAIT routine, 2–5, 2–11, 2–15
EXE_STD$KP_STARTIO routine, 1–2, 2–11, 2–15
EXE_STD$LCLDSKVALID, 5–4
EXE_STD$LCLDSKVALID routine, 2–3, 2–11, 2–15
EXE_STD$MAXACMODE routine, 2–11
EXE_STD$MNTVERSIO routine, 2–11
EXE_STD$MODIFY, 5–5
EXE_STD$MODIFY routine, 2–3, 2–11, 2–15
EXE_STD$MODIFYLOCK routine, 2–6, 2–11, 2–15
EXE_STD$MOUNT_VER routine, 2–11
EXE_STD$ONEPARM, 5–5
EXE_STD$ONEPARM routine, 2–3, 2–11, 2–15
EXE_STD$OUTZSTRING routine, 2–12
EXE_STD$PRIMITIVE_FORK routine, 2–12
EXE_STD$PRIMITIVE_FORK_WAIT routine, 2–12

EXE_STD$PROBER routine, 2–12
EXE_STD$PROBER_DSC routine, 2–12
EXE_STD$QIOACPPKT routine, 2–5, 2–12, 2–15
EXE_STD$QIODRVPKT, 5–5
EXE_STD$QIODRVPKT routine, 2–5, 2–12, 2–15
EXE_STD$QUEUE_FORK routine, 2–12
EXE_STD$QXQPPKT routine, 2–12
EXE_STD$READ, 5–5
EXE_STD$READ routine, 2–3, 2–12, 2–15
EXE_STD$READCHK routine, 2–6, 2–12, 2–15
EXE_STD$READLOCK routine, 2–6, 2–12, 2–15
EXE_STD$READLOCK_ERR routine, 2–15
EXE_STD$RMVTIMQ routine, 2–12
EXE_STD$SENSEMODE, 5–5
EXE_STD$SENSEMODE routine, 2–3, 2–12, 2–16
EXE_STD$SETCHAR, 5–5
EXE_STD$SETCHAR routine, 2–3, 2–12, 2–16
EXE_STD$SETMODE, 5–5
EXE_STD$SETMODE routine, 2–3, 2–12, 2–16
EXE_STD$SNDEVMSG routine, 2–12
EXE_STD$SNGLEQUOTA_LONG routine, 2–12
EXE_STD$STARTIO routine, 2–1
EXE_STD$SYNCH_LOOP routine, 2–12
EXE_STD$WRITE, 5–5
EXE_STD$WRITE routine, 2–3, 2–12, 2–16
EXE_STD$WRITECHK routine, 2–6, 2–12, 2–16
EXE_STD$WRITELOCK routine, 2–6, 2–12, 2–16
EXE_STD$WRTMAILBOX routine, 2–12
EXE_STD$ZEROPARM, 5–5
EXE_STD$ZEROPARM routine, 2–3, 2–12, 2–16

F

FDT (function decision table)
  address, 4–3
  creating, 4–4 to 4–9
  defining, 2–2
FDT action routine vector, 4–4
FDT completion routine, 5–1
FDT completion routines, 2–4, 5–6 to 5–8
FDT exit routines, 2–5
FDT processing
  calling sequence, 5–2 to 5–4
FDT routine
  allocating system buffer in, 5–9
  context, 5–2
  creating, 5–1 to 5–11
  for buffered I/O, 5–8 to 5–11
  for direct I/O, 5–5, 5–8
  register usage, 5–2
  system-provided, 5–4 to 5–11
  upper-level, 4–6, 5–1, 5–2 to 5–4
FDT routines
  upper-level, 2–2
FDT support routine, 5–1
FDT support routines, 2–5
$FDTARGDEF macro, 2–4
FDT_ACT macro, 2–2, 4–6
FDT_BUF macro, 2–2, 4–5
FDT_CONTEXT structure, 2–5
FDT_INI macro, 2–2, 4–5
FORK macro, 1–1
Fork routines, 1–1
FORK_ROUTINE macro, 1–1
FORK_WAIT macro, 1–1
Full duplex device driver, 5–7
FUNCTAB macro, 2–2

G

Guidelines for converting Step 1 drivers to Step 2, 2–1 to 2–16

I

I/O database
  creation, 4–1
I/O function
  buffered, 2–2
  indicating a buffered, 4–4
  indicating as legal to a device, 4–4
  legal, 2–2, 4–4
I/O function code
  defining device-specific, 4–9
  system-defined, 4–6 to 4–9
I/O postprocessing
  device-dependent, 5–9
  device-independent, 5–10
  for buffered I/O, 5–9 to 5–11
I/O preprocessing
  completing, 4–6
  device-dependent, 5–1 to 5–11
I/O request
  synchronizing simultaneous processing of multiple, 5–7
  with no parameters, 5–5
  with one parameter, 5–5
IO$_AVAILABLE function, 5–4
IO$_PACKACK function, 5–4
IO$_UNLOAD function, 5–4
IOC$CANCELIO routine, 2–1
IOC$CLONE_UCB routine, 2–16
IOC$LINK_UCB routine, 2–16
IOC$MNTVER routine, 2–1
IOC_STD$ALLOC_CNT_RES routine, 2–12
IOC_STD$ALTREQCOM routine, 2–12
IOC_STD$BROADCAST routine, 2–12
IOC_STD$CANCELIO routine, 2–1, 2–12
IOC_STD$CLONE_UCB routine, 2–12, 2–16
IOC_STD$COPY_UCB routine, 2–12
IOC_STD$CREDIT_UCB routine, 2–12

IOC_STD$CVTLOGPHY routine, 2–13
IOC_STD$CVT_DEVNAM routine, 2–13
IOC_STD$DELETE_UCB routine, 2–13
IOC_STD$DIAGBUFILL routine, 2–13
IOC_STD$FILSPT routine, 2–13
IOC_STD$GETBYTE routine, 2–13
IOC_STD$INITBUFWIND routine, 2–13
IOC_STD$INITIATE routine, 2–13
IOC_STD$LINK_UCB routine, 2–13, 2–16
IOC_STD$MAPVBLK routine, 2–13
IOC_STD$MNTVER routine, 2–1, 2–13
IOC_STD$MOVFRUSER routine, 2–13
IOC_STD$MOVFRUSER2 routine, 2–13
IOC_STD$MOVTOUSER routine, 2–13
IOC_STD$MOVTOUSER2 routine, 2–13
IOC_STD$PARSDEVNAM routine, 2–13
IOC_STD$POST_IRP routine, 2–13
IOC_STD$PRIMITIVE_REQCHANH routine, 2–13
IOC_STD$PRIMITIVE_REQCHANL routine, 2–13
IOC_STD$PRIMITIVE_WFIKPCH routine, 2–13
IOC_STD$PRIMITIVE_WFIRLCH routine, 2–13
IOC_STD$PTETOPFN routine, 2–13
IOC_STD$QNXTSEG1 routine, 2–13
IOC_STD$RELCHAN routine, 2–13
IOC_STD$REQCOM routine, 2–13
IOC_STD$SEARCHDEV routine, 2–13
IOC_STD$SEARCHINT routine, 2–13
IOC_STD$SEVER_UCB routine, 2–13
IOC_STD$SIMREQCOM routine, 2–13
IOC_STD$THREADCRB routine, 2–13
$IODEF macro, 4–6
IOFORK macro, 1–1
$IOUNLOCK macro, 2–14
IRP$L_BCNT
  writing, 5–9
IRP$L_BOFF, 5–9, 5–10
IRP$L_STS
  for read function, 5–9
IRP$L_SVAPTE
  for buffered I/O, 5–9, 5–10
IRP$V_FUNC, 5–9, 5–10
IRP$W_STS
  for read function, 5–10
  for write function, 5–10

J

JIB$L_BYTCNT, 5–9, 5–10
JSB replacement macro, 1–2
JSB-based system routine
  naming, 1–2

K

Kernel process, 1–1

L

LDR_STD$ALLOC_PT routine, 2–13
LDR_STD$DEALLOC_PT routine, 2–13
Legal I/O function, 2–2
LNM_STD$DELETE_LNMB routine, 2–13
LNM_STD$LOCKW routine, 2–13
LNM_STD$UNLOCK routine, 2–13

M

Macro
  JSB replacement, 1–2
MMG_STD$ALLOC_PFN routine, 2–13
MMG_STD$DALLOC_PFN routine, 2–14
MMG_STD$IOLOCK routine, 2–14
MMG_STD$SVAPTECHK routine, 2–14
MMG_STD$UNLOCK routine, 2–14
Modify function
  FDT routine for, 5–5
Mount verification routine
  address, 4–4
MT$CHECK_ACCESS routine, 2–3, 2–16
MT_STD$CHECK_ACCESS routine, 2–3, 2–14, 2–16

P

PCB$L_JIB, 5–9
Performance, 3–1 to 3–2
PIO transfer
  using buffered I/O in, 4–5
PMS_STD$ABORT_RQ routine, 2–14
PMS_STD$START_RQ routine, 2–14
Process context, 5–2
Process quota
  byte count, 5–10

R

Read function
  FDT routine for, 5–5
$READLOCK_ERR macro, 2–15
Register dumping routine
  address, 4–3
REQCHAN macro, 1–1

S

SCH_STD$CLREF routine, 2–14
SCH_STD$IOLOCKR routine, 2–14

SCH_STD$IOLOCKW routine, 2–14
SCH_STD$IOUNLOCK routine, 2–14
SCH_STD$LOCKW routine, 2–14
SCH_STD$LOCKWEXEC routine, 2–14
SCH_STD$LOCKWEXEC_QUAD routine, 2–14
SCH_STD$POSTEF routine, 2–14
SCH_STD$QAST routine, 2–14
SCH_STD$RAVAIL routine, 2–14
SCH_STD$UNLOCK routine, 2–14
SCH_STD$UNLOCKEXEC routine, 2–14
SCH_STD$UNLOCKEXEC_QUAD routine, 2–14
SCH_STD$WAKE routine, 2–14
Sense device characteristics function, 5–5
Sense device mode function, 5–5
Set device characteristics function, 5–5
Set device mode function, 5–5
SMP_STD$ACQNOIPL routine, 2–14
SMP_STD$ACQUIRE routine, 2–14
SMP_STD$ACQUIREL routine, 2–14
SMP_STD$RELEASE routine, 2–14
SMP_STD$RELEASEL routine, 2–14
SMP_STD$RESTORE routine, 2–14
SMP_STD$RESTOREL routine, 2–14
SS$_FDT_COMPL status, 2–4, 2–5
Start I/O routine
  address, 4–3
Start-I/O routine
  kernel process, 4–4
Step 1 device driver, 1–1
Step 2 device driver, 1–1
  optimizing, 3–1 to 3–2
System page-table entry
  allocating permanent, 4–2

U

UCB (unit control block)
  number to be created, 4–2
UCB$V_BSY, 5–7
Unit delivery routine
  address, 4–2
Unit initialization routine
  address, 4–4
Upper-level FDT action routine, 5–1, 5–2 to 5–4
  identifying, 4–6
Upper-level FDT action routines
  defining, 2–2

W

WFIKPCH macro, 1–1
WFIRLCH macro, 1–1
Write function
  FDT routine for, 5–5