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

Order Number: AA–Q28TA–TE

March 1994

This manual describes how to convert an OpenVMS AXP Step 1 device driver, written in VAX MACRO, to an OpenVMS AXP Step 2 driver, also written in VAX MACRO.

Revision/Update Information: This is a new manual.

Software Version: OpenVMS AXP Version 6.1

Digital Equipment Corporation
Maynard, Massachusetts

March 1994

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

This document is available on CD–ROM.

ZK6321

This document was prepared using VAX DOCUMENT Version 2.1.

Send Us Your Comments

We welcome your comments on this or any other OpenVMS manual. If you have suggestions for improving a particular section or find any errors, please indicate the title, order number, chapter, section, and page number (if available). We also welcome more general comments. Your input is valuable in improving future releases of our documentation.

You can send comments to us in the following ways:

- Internet electronic mail: OPENVMSDOC@ZKO.MTS.DEC.COM
- Fax: 603-881-0120 Attn: OpenVMS Documentation, ZKO3-4/U08
- A completed Reader's Comments form (postage paid, if mailed in the United States), or a letter, via the postal service. Two Reader's Comments forms are located at the back of each printed OpenVMS manual. Please send letters and forms to:

  Digital Equipment Corporation
  Information Design and Consulting
  OpenVMS Documentation
  110 Spit Brook Road, ZKO3-4/U08
  Nashua, NH 03062-2698
  USA

You may also use an online questionnaire to give us feedback. Print or edit the online file SYS$HELP:OPENVMSDOC_SURVEY.TXT. Send the completed online file by electronic mail to our Internet address, or send the completed hardcopy survey by fax or through the postal service.

Thank you.

Contents

Index

Tables

Preface

This manual describes how to convert an OpenVMS AXP Step 1 driver to an OpenVMS AXP Step 2 driver. It explains how you must change Step 1 driver code to prepare the driver to be compiled, linked, loaded, and run as a Step 2 driver. This manual highlights specific changes that you must make to driver routines and tables.

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 the OpenVMS AXP operating system, Version 6.1.

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 presents an overview of the new Step 2 device driver interfaces.
- Chapter 2 contains guidelines for converting a Step 1 device driver to a Step 2 device driver.
- Chapter 3 provides tips for converting complex or unusual drivers.

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 *OpenVMS AXP Device Support: Reference*. For basic information about the components of OpenVMS device drivers and OpenVMS requirements for them, refer to the following manuals:

- *OpenVMS AXP Device Support: Developer's Guide*
- *OpenVMS AXP Device Support: Reference*

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*
- *OpenVMS 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 an 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 6.1 introduces formal support for user-written device drivers and a new device driver interface known as the **Step 2** driver interface. The Step 2 driver interface replaces the temporary Step 1 driver interface that was provided in OpenVMS AXP Versions 1.0 and 1.5. As a result, if you are supplying a driver to run under OpenVMS AXP Version 6.1, it must comply with the **Step 2** driver interfaces described in this manual.

Although the Step 2 driver interfaces allow you to write OpenVMS AXP device drivers in high-level languages, a Step 2 device driver can also be written in VAX MACRO. The guidelines in this manual describe how to convert a Step 1 driver written in VAX MACRO to a Step 2 driver written in VAX MACRO.

1.1 Overview of Step 2 Driver Changes

The key difference between the Step 1 and Step 2 interfaces is the use of standard call interfaces, which replace the JSB interfaces used throughout Step 1 drivers. Standard call interfaces are now required for the following driver-supplied routines:

- Cancel I/O routine
- Cancel selective routine
- Channel assign routine
- Cloned UCB routine
- Controller initialization routine
- Function decision table (FDT) routines
- Mount verification routine
- Register dumping routine
- Unit delivery routine
- Unit initialization routine

Standard call interfaces are optional for the following driver-supplied routines:

- Alternate start I/O routine
- Start I/O routine
- Driver fork routines

Additional Step 2 changes include the following:

- Function decision table processing does not rely on the RET under JSB mechanism.

- The layout of the function decision table is significantly different.
- Standard call interfaces are available for most OpenVMS support routines.
- A small number of OpenVMS support routines with JSB interfaces are no longer available.

For detailed information about these changes, see Chapter 2.

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 Overview of Step 1 and Step 2 Driver Similarities

Step 2 drivers are similar to Step 1 drivers in the following ways:

- The overall structure of a device driver is unchanged.
- Interfaces for internal driver routines are not dictated.
- Interfaces between port and class drivers are not dictated.
- JSB interfaces continue to be available for most OpenVMS support routines used by drivers.
- The use of the kernel process mechanism is essentially unchanged.
- The Start I/O to REQCOM code path can use JSB interfaces.

1.3 Step 2 Driver Naming Conventions

The following naming conventions apply to the new call-based system routines:

- 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. For example, EXE_STD$FINISHIO is the call-based routine that replaces the JSB-based EXE$FINISHIO.
- If a JSB-replacement macro exists for x$y, it is named CALL_Y.

  For example, you can replace a JSB to EXE$FINISHIO with the CALL_FINISHIO macro. CALL_FINISHIO issues a standard call to EXE_STD$FINISHIO after loading the standard call argument registers from the general registers used in the traditional JSB to EXE$FINISHIO.
- When using the call-based system routine directly, note 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 parameter 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 *OpenVMS AXP Device Support: Reference*.

1.4 Converting Drivers Written in BLISS

This manual focuses on converting existing Step 1 device drivers, written in VAX MACRO, to Step 2 device drivers. However, the call interfaces described are equally available to Step 1 drivers written in BLISS. To convert a Step 1 BLISS driver, remove the JSB linkages from routine declarations and verify the specified parameter order for any given routine against that listed in the system routines section of *OpenVMS AXP Device Support: Reference*.

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. You must convert these VAX MACRO modules as described in this manual. Alternatively, you can now use new BLISS macros that allow you to code the DPT, DDT, and FDT declarations in BLISS. For more information about these macros, see *OpenVMS AXP Device Support: Reference*.

1.5 Writing Step 2 Drivers in C

OpenVMS AXP Version 6.1 provides the support necessary to write a device driver in the C programming language. For information about writing OpenVMS AXP device drivers in the C programming language or another high-level language, see the *OpenVMS AXP Device Support: Developer's Guide*.

2
Conversion Guidelines

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

2.1 DPTAB Changes

The driver prologue table (DPT) must declare that the driver is a Step 2 driver. In the driver's DPTAB macro invocation, replace **step**= **1** with **step**=**2**. For example:

```
Step 1                          Step 2

DPTAB -                         DPTAB   -
   STEP = 1,-                     STEP = 2,-
```

If you do not make this change, compilation errors will result. See *OpenVMS AXP Device Support: Reference* for more information about the DPT and the DPTAB macro.

2.2 DDTAB Changes

The routines pointed to by the driver dispatch table (DDT) must conform to Step 2 requirements. You may need to change entry point declarations for driver-specific routines, but the names may remain unchanged. Any OpenVMS routine names referenced should be changed as follows in the driver's DDTAB macro invocation:

1. Replace **cancel**=IOC$CANCELIO with **cancel**=IOC_STD$CANCELIO.
2. Replace **mntver**=IOC$MNTVER with **mntver**=IOC_STD$MNTVER.
3. Replace **start**=EXE$KP_STARTIO with **start**=EXE_STD$KP_STARTIO.

See *OpenVMS AXP Device Support: Reference* for more information about the driver dispatch table (DDT) and the DDTAB macro.

For complex Start I/O to REQCOM code paths that use extensive branching between .JSB_ENTRY routines, you can use the traditional Step 1 JSB interface for the start I/O and the alternate start I/O routines. For more information, see Section 2.6.1.

2.3 Driver Entry Point Routine Changes

To replace the JSB interfaces used throughout Step 1 drivers with the call interfaces required for Step 2 driver-supplied routines, perform the following tasks:

1. 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 the following:
   - $DRIVER_CANCEL_ENTRY
   - $DRIVER_CANCEL_SELECTIVE_ENTRY
   - $DRIVER_CHANNEL_ASSIGN_ENTRY
   - $DRIVER_CLONEDUCB_ENTRY
   - $DRIVER_CTRLINIT_ENTRY
   - $DRIVER_ERRRTN_ENTRY
   - $DRIVER_FDT_ENTRY
   - $DRIVER_MNTVER_ENTRY
   - $DRIVER_REGDUMP_ENTRY
   - $DRIVER_DELIVER_ENTRY
   - $DRIVER_UNITINIT_ENTRY
2. Use the default **FETCH=YES** parameter value.

   This value causes the standard interface parameters to be fetched and copied to their traditional JSB interface registers, for example:

   ```
   $DRIVER_UNITINIT_ENTRY FETCH=YES
   ```

   results in

   ```
   MOVL  #SS$_NORMAL,R0
   MOVL  UNITARG$_IDB(AP),R4
   MOVL  UNITARG$_UCB(AP),R5
   ```
3. Use the default **PRESERVE** parameter value.

   The default is the set of registers that was allowed to be scratched by the Step 1 JSB interface routine, for example:

   ```
   $DRIVER_UNITINIT_ENTRY
   ```

   results in

   PRESERVE=<R2>

   This set of registers is augmented by the MACRO-32 compiler register autopreservation feature. Use the **.SET_REGISTERS WRITTEN=<Rn>** directive to augment this set of registers manually.
4. Make sure that each Step 2 driver routine returns control to the operating system with a RET instruction, instead of an RSB instruction.

See *OpenVMS AXP Device Support: Reference* for more information about the Step 2 driver entry point macros.

2.4 FUNCTAB Macro Changes

A Step 1 driver contains three or more FUNCTAB macro invocations. For Step 2 drivers, the function decision table (FDT) format is significantly different. Step 2 driver changes include the following:

- The FUNCTAB macro is obsolete.

- The FDT structure consists of a 64-bit mask specifying the buffered functions and a 64-entry vector pointing to the upper-level FDT action routine that corresponds to each of the I/O function codes. There is no bit mask of legal functions.
- Three new macros are used to build the FDT:

  FDT_INI initializes an FDT structure
  FDT_BUF declares the buffered I/O functions
  FDT_ACT declares an upper-level FDT action routine for a set of I/O functions

You must make the following changes:

1. Delete the first FUNCTAB macro, the one that identifies valid I/O function codes, and the 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.
2. Replace the second FUNCTAB macro, the one that identifies buffered I/O functions, with an FDT_BUF macro. Replace the word "FUNCTAB" with the word "FDT_BUF" and remove the first null argument.
3. Replace each subsequent FUNCTAB macro with an FDT_ACT macro.

   For example:

```
Step 1 FDT Declaration

MY_FUNCTBL:

  FUNCTAB ,-      ;legal func
          <SENSEMODE,SENSECHAR,-
           WRITELBLK,WRITEPBLK>

  FUNCTAB ,-      ;buffered func
          <SENSEMODE,SENSECHAR>

  FUNCTAB EXE$SENSE_MODE,-
              <SENSEMODE,SENSECHAR>

  FUNCTAB MY_FDT_WRITE,-
              <WRITELBLK,WRITEPBLK>


Step 2 FDT Declaration

  FDT_INI MY_FUNCTBL

  FDT_BUF <SENSEMODE,SENSECHAR>

  FDT_ACT EXE_STD$SENSE_MODE,-
              <SENSEMODE,SENSECHAR>

  FDT_ACT MY_FDT_WRITE,-
              <WRITELBLK,WRITEPBLK>
```

Because Step 2 driver support replaces all system-supplied upper-level FDT action 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

Obsolete Step 1 Routine	Step 2 FDT Action Routine
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
New for Step 2	EXE$ILLIOFUNC
EXE$LCLDSKVALID	EXE_STD$LCLDSKVALID
EXE$MODIFY	EXE_STD$MODIFY
EXE$ONEPARM	EXE_STD$ONEPARM
EXE$READ	EXE_STD$READ
EXE$SENSEMODE	EXE_STD$SENSEMODE
EXE$SETCHAR	EXE_STD$SETCHAR
EXE$SETMODE	EXE_STD$SETMODE
EXE$WRITE	EXE_STD$WRITE
EXE$ZEROPARM	EXE_STD$ZEROPARM
MT$CHECK_ACCESS[1]	MT_STD$CHECK_ACCESS

[1] For information about changes in routine behavior, see *OpenVMS AXP Device Support: Reference.*

For more information about the FDT_INI, FDT_BUF, and FDT_ACT macros and the upper-level FDT action, see *OpenVMS AXP Device Support: Reference.*

Warning

Step 2 device drivers support only a single upper-level FDT action routine per I/O function code. 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. For more information about composite routines, see Chapter 3.

2.5 FDT Routine Changes

The Step 2 FDT routine changes you need to make depend on the type of FDT routine your driver includes. This section names and describes types of FDT routines, summarizes the differences between Step 1 and Step 2 FDT processing, and specifies the required Step 2 FDT routine changes.

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. 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 *OpenVMS AXP Device Support: Reference* for a full description of the formal interface to an upper-level FDT action routine.)

OpenVMS provides a set of upper-level FDT action routines, but drivers can also define their own driver-specific upper-level FDT action routines. EXE_ STD$READ is an example of a Step 2 upper-level FDT action routine.

An **FDT exit routine** is a routine used by a Step 1 driver to terminate FDT processing and exit from the $QIO system service. For example, EXE$QIODRVPKT is an FDT exit routine. FDT exit routines use the **RET-under-JSB** mechanism to exit from the $QIO system service. The RET under JSB mechanism is the technique of using a RET instruction to return from a JSB interface routine. This RET instruction causes control to return from the most recent CALL interface routine on the current call tree. This technique unwinds any intervening JSB interface routines without returning to their callers and without restoring any register values that were saved by the unwound JSB routines. In a Step 2 driver, FDT exit routines have been replaced by FDT completion routines.

FDT completion routines are the Step 2 replacements for Step 1 FDT exit routines. Like FDT exit routines, completion routines complete FDT processing by queuing the I/O request to the appropriate next stage of processing. Unlike FDT exit routines, FDT completion routines return back to their callers and do not rely on the RET-under-JSB mechanism. EXE_STD$QIODRKPT is an example of a Step 2 FDT exit routine.

FDT support routines are routines that are called during FDT processing, but they are not upper-level FDT action routines. They have code paths that call FDT completion routines, but they do not complete FDT processing themselves. Step 1 FDT support routines must use a JSB interface. OpenVMS provides a set of FDT support routines, but drivers can also include their own support routines. EXE_STD$READCHK is an example of a Step 2 FDT support routine.

For Step 1 drivers:

- Upper-level FDT action routines are invoked via a JSB interface.
- A return from an upper-level FDT action routine via an RSB instruction returns control back to the FDT dispatch loop.
- FDT support routines are all invoked via a JSB interface.
- Exit from Step 1 FDT processing and the QIO system service is via a RET under JSB in an FDT exit routine, for example, EXE$ABORTIO, EXE$QIODRVPKT, and so on.

In contrast, for Step 2 drivers:

- Upper-level FDT action routines are invoked via a new standard call interface.
- Control is returned from an upper-level FDT action routine via a RET instruction, which exits the FDT dispatcher and returns to the $QIO system service.
- Driver-specific FDT support routines may continue to use JSB interfaces, however OpenVMS-provided FDT support routines should be invoked using the new CALL_x macros.
- FDT completion routines are used instead of FDT exit routines. FDT completion routines return back to their callers with the SS$_FDT_COMPL status. All upper-level FDT action routines must return this status back to the $QIO system service.

2.5.1 Upper-Level Routine Entry Point Changes

If the Step 1 driver you are converting to Step 2 includes a device-specific upper-level FDT action routine, perform the following tasks:

1. Replace the .JSB_ENTRY MACRO-32 compiler directive used to define the FDT routine entry point with a $DRIVER_FDT_ENTRY macro. (See *OpenVMS AXP Device Support: Reference*.) This macro declares the routine's call entry point and ensures, by default, that all nonscratch registers defined by the OpenVMS Calling Standard are preserved. This macro also invokes the $FDTARGDEF macro, thus allowing the FDT routine to access its arguments at their standard locations with respect to the AP.
2. 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_FDT_2. Note that the Step 2 FDT format differs from the Step 1 format.
3. Use the default register PRESERVE list on $DRIVER_FDT_ENTRY macro.

2.5.2 FDT Exit Routine Changes

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. Use the default value for the **do_ret**=**YES** parameter.

For example, replace either:

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

or:

```
JSB G^EXE$ABORTIO
RSB
```

with:

```
CALL_ABORTIO
```

Table 2–2 FDT Completion Routines and Macros

Obsolete Step 1 FDT Exit Routine	Macro	FDT Completion Routine
EXE$ABORTIO	CALL_ABORTIO	EXE_STD$ABORTIO
EXE$ALTQUEPKT	CALL_ALTQUEPKT[1]	EXE_STD$ALTQUEPKT
EXE$FINISHIO	CALL_FINISHIO	EXE_STD$FINISHIO
EXE$FINISHIOC	CALL_FINISHIOC	EXE_STD$FINISHIO
New for Step 2	CALL_FINISHIO_NOIOST	EXE_STD$FINISHIO
EXE$IORSNWAIT	CALL_IORSNWAIT	EXE_STD$IORSNWAIT
EXE$QIOACPPKT	CALL_QIOACPPKT	EXE_STD$QIOACPPKT
EXE$QIODRVPKT	CALL_QIODRVPKT	EXE_STD$QIODRVPKT
EXE$QIORETURN	none	none[2]

[1]The CALL_ALTQUEPKT macro does not provide the **do_ret** argument. An FDT routine that invokes CALL_ALTQUEPKT must typically manage the dispatching of I/O requests to the driver's alternate start-I/O entry point.

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

a. Releases the device lock if held. EXE$QIORETURN contained code that unconditionally released the device lock.
b. Places SS$_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, your driver need do nothing special to deliver a success status to the $QIO caller.

If you call an FDT completion routine directly (that is, not using 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 *OpenVMS AXP Device Support: Reference* for more information about FDT completion routines and a detailed description of the macros.

2.5.3 OpenVMS-Supplied FDT Support Routine Changes

For Step 2 drivers, replace any JSB instruction to an OpenVMS supplied FDT support routine with the appropriate JSB-replacement macro. (See Table 2–3.) The macros do the following:

- 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.

- Restore the output values into the output registers for the corresponding Step 1 routine.
- Generate code that checks the returned status and invokes a RET instruction on an error. (Some Step 1 FDT support routines never returned to their callers in the event of an error.)

Table 2–3 System-Supplied FDT Support Routines

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

See *OpenVMS AXP Device Support: Reference* for further discussion of system-supplied FDT support routines and details about the macros.

2.5.4 Driver-Supplied FDT Support Routine Changes

It is easiest to use your current JSB interfaces for all driver-supplied FDT support routines. In fact, the correct operation of the CALL_x macros depends on keeping the JSB interfaces for your support routines.

To convert a Step 1 driver that contains driver-supplied FDT support routines to the Step 2 interface, do the following:

1. Use the $DRIVER_FDT_ENTRY macro for upper-level routines with the default preserve list, regardless of the registers that are actually modified by the upper-level FDT routine.
2. Use the FDT completion macros with DO_ RET=YES (the default) and the FDT support routines in Table 2–3.
3. Keep the JSB interface for all driver-supplied FDT support routines.

If you want to convert driver-supplied FDT support routines to CALL interfaces, see Chapter 3.

2.5.5 Returning from Upper-Level Routines

In most cases, upper-level FDT action routines end with a call to an FDT completion macro that includes a RET instruction. However, if after following the steps outlined in Section 2.5.1 through Section 2.5.4, you still have an RSB instruction in your upper-level FDT action routine, you should change it to the following:

```
MOVL #SS$_NORMAL,R0
RET
```

Encountering an RSB instruction in your upper-level FDT action routine indicates that the upper-level FDT action routine, which you are converting, is one of several upper-level routines called for a single I/O function. Because Step 2 drivers can have only one upper-level FDT action routine for each I/O function, you must also make this FDT routine a composite FDT routine. For information about composite FDT routines, see Section 3.1.

2.6 Start I/O to REQCOM Changes

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.

- If your start I/O code path is simple, you can convert it to the new standard call interface as described in Chapter 3. You may be able to convert some fork routines independently to call interfaces, for example, for routines queued from a unit initialization routine.
- If your start I/O code path is more complicated, you might want to use the JSB interfaces described in Section 2.6.1.

The Step 1 kernel process mechanism uses a standard call interface. For a Step 2 device driver that uses the kernel process mechanism, the path from start I/O to request complete remains essentially unchanged. Section 2.6.2 describes the two changes necessary for Step 2 drivers.

2.6.1 Simple Fork Mechanism—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 can 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. 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 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.

If you continue using JSB-based fork routines, 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.

2.6.2 Kernel Process Mechanism

Device drivers that use the kernel process mechanism require two minor changes to the driver code paths from start I/O through request complete.

1. Because of the new start I/O interface, these drivers 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
```

2. A device interrupt service routine can use one of two methods to resume a kernel process thread that has been suspended by IOC$KP_WFIKPCH or IOC$KP_WFIRLCH. The first and preferred method is to call EXE$KP_RESTART. This method is unchanged for Step 2 drivers.

   The second method is to load R3 and R4 from the UCB fork block and then invoke the routine whose procedure value is in UCB$L_FPC(R5).

   In a Step 2 driver, this is done using a standard call interface, as follows:

```
PUSHL   R5              ;P3 = UCB address
PUSHL   UCB$Q_FR4(R5)   ;P2 = FR4 value
PUSHL   UCB$Q_FR3(R5)   ;P1 = FR3 value
CALLS   #3,@UCB$L_FPC(R5)
```

   instead of the following:

```
MOVL    UCB$Q_FR3(R5),R3
MOVL    UCB$Q_FR4(R5),R4
JSB     @UCB$L_FPC(R5)
```

2.7 Common OpenVMS-Supplied EXEC Routines

Replace any JSB to the routines listed in Table 2–4 with the appropriate macro. If the interface provided by the JSB-replacement macro differs from the original JSB interface, the macro generates a compile-time warning. The compile-time warning identifies the register output that is not provided by the replacement macro. After you have made sure that your code does not depend on this output you can disable the warning by using the INTERFACE_WARNING=NO parameter on the macro.

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 *OpenVMS AXP Device Support: Reference* for a detailed description of the macros.

Most of the JSB-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 Device Support: Reference* to verify the routine interface. You can optimize performance of the macro by following the recommendations listed in Chapter 3.

Table 2–4 Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
ACP$ACCESS[1]	CALL_ACCESS	No	No
ACP$ACCESSNET[1]	CALL_ACCESSNET	No	No
ACP$DEACCESS[1]	CALL_DEACCESS	No	No
ACP$MODIFY[1]	CALL_ACP_MODIFY	No	No
ACP$MOUNT[1]	CALL_MOUNT	No	No
ACP$READBLK[1]	CALL_READBLK	No	No
ACP$WRITEBLK[1]	CALL_WRITEBLK	No	No
COM$DELATTNAST	CALL_DELATTNAST	No	No
COM$DELATTNASTP	CALL_DELATTNASTP	No	No
COM$DELCTRLAST	CALL_DELCTRLAST	No	No
COM$DELCTRLASTP	CALL_DELCTRLASTP	No	No
COM$DRVDEALMEM	CALL_DRVDEALMEM	No	No
COM$FLUSHATTNS	CALL_FLUSHATTNS	No	No
COM$FLUSHCTRLS	CALL_FLUSHCTRLS	No	No
COM$POST	CALL_POST	No	No
COM$POST_NOCNT	CALL_POST_NOCNT	No	No
COM$SETATTNAST[1]	CALL_SETATTNAST	No	No
COM$SETCTRLAST[1]	CALL_SETCTRLAST	No	No
ERL$ALLOCEMB	CALL_ALLOCEMB	No	No

[1]The JSB-based Step 1 routine is not supported by the OpenVMS AXP operating system Version 6.1.

(continued on next page)

Table 2–4 (Cont.) Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
ERL$DEVICEATTN	CALL_DEVICEATTN	No	No
ERL$DEVICERR	CALL_DEVICERR	No	No
ERL$DEVICTMO	CALL_DEVICTMO	No	No
ERL$RELEASEMB	CALL_RELEASEMB	No	No
EXE$ABORTIO[1]	CALL_ABORTIO	No	No
EXE$ALLOCBUF	CALL_ALLOCBUF	No	No
EXE$ALLOCIRP	CALL_ALLOCIRP	No	No
EXE$ALTQUEPKT	CALL_ALTQUEPKT	No	No
EXE$CARRIAGE	CALL_CARRIAGE	No	No
EXE$CHKCREACCES	CALL_CHKCREACCES	No	R1
EXE$CHKDELACCES	CALL_CHKDELACCES	No	R1
EXE$CHKEXEACCES	CALL_CHKEXEACCES	No	R1
EXE$CHKLOGACCES	CALL_CHKLOGACCES	No	R1
EXE$CHKPHYACCES	CALL_CHKPHYACCES	No	R1
EXE$CHKRDACCES	CALL_CHKRDACCES	No	R1
EXE$CHKWRTACCES	CALL_CHKWRTACCES	No	R1
EXE$FINISHIO[1]	CALL_FINISHIO	No	No
EXE$FINISHIOC[1]	CALL_FINISHIOC	No	No
EXE$INSERT_IRP	CALL_INSERT_IRP	No	No
EXE$INSIOQ	CALL_INSIOQ	No	No
EXE$INSIOQC	CALL_INSIOQC	No	No
EXE$IORSNWAIT[1]	CALL_IORSNWAIT	No	No
EXE$LCLDSKVALID[1]	CALL_LCLDSKVALID	No	No
EXE$MNTVERSIO	CALL_MNTVERSIO	No	No
EXE$MODIFY[1]	CALL_EXE_MODIFY	No	No
EXE$MODIFYLOCK[1]	CALL_MODIFYLOCK	No	No
EXE$MODIFYLOCK_ERR[1]	CALL_MODIFYLOCK_ ERR	Yes	No
EXE$MOUNT_VER	CALL_MOUNT_VER	No	R0 and R1
EXE$ONEPARM[1]	CALL_ONEPARM	No	No
EXE$PRIMITIVE_FORK	FORK[2]	No	No
EXE$PRIMITIVE_FORK_WAIT	FORK_WAIT[2]	No	No
EXE$QIOACPPKT[1]	CALL_QIOACPPKT	No	No
EXE$QIODRVPKT[1]	CALL_QIODRVPKT	No	No
EXE$QXQPPKT[1]	CALL_QXQPPKT	No	No
EXE$READCHK [1]	CALL_READCHK	No	No

[1]The JSB-based Step 1 routine is not supported by the OpenVMS AXP operating system Version 6.1.

[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.) Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
EXE$READCHKR[1]	CALL_READCHKR	No	No
EXE$READLOCK[1]	CALL_READLOCK	No	No
EXE$READLOCK_ERR[1]	CALL_READLOCK_ ERR	Yes	No
EXE$SENSEMODE[1]	CALL_SENSEMODE	No	No
EXE$SETCHAR[1]	CALL_SETCHAR	No	No
EXE$SETMODE[1]	CALL_SETMODE	No	No
EXE$SNDEVMSG	CALL_SNDEVMSG	No	No
EXE$WRITE[1]	CALL_WRITE	No	No
EXE$WRITECHK[1]	CALL_WRITECHK	No	No
EXE$WRITECHKR[1]	CALL_WRITECHKR	No	No
EXE$WRITELOCK[1]	CALL_WRITELOCK	No	No
EXE$WRITELOCK_ERR[1]	CALL_WRITELOCK_ ERR	Yes	No
EXE$WRTMAILBOX	CALL_WRTMAILBOX	No	No
EXE$ZEROPARM[1]	CALL_ZEROPARM	No	No
IOC$ALTREQCOM	CALL_ALTREQCOM	No	No
IOC$BROADCAST	CALL_BROADCAST	No	R1
IOC$CANCELIO	CALL_CANCELIO	No	R0 and R1
IOC$CLONE_UCB[1]	CALL_CLONE_UCB	Yes	No
IOC$COPY_UCB	CALL_COPY_UCB	No	No
IOC$CREDIT_UCB	CALL_CREDIT_UCB	No	No
IOC$CVTLOGPHY	CALL_CVTLOGPHY	No	No
IOC$CVT_DEVNAM	CALL_CVT_DEVNAM	No	No
IOC$DELETE_UCB	CALL_DELETE_UCB	No	No
IOC$DIAGBUFILL	CALL_DIAGBUFILL	No	No
IOC$FILSPT	CALL_FILSPT	No	No
IOC$GETBYTE	CALL_GETBYTE	No	No
IOC$INITBUFWIND	CALL_INITBUFWIND	No	No
IOC$INITIATE	CALL_INITIATE	No	No
IOC$LINK_UCB[1]	CALL_LINK_UCB	Yes	No
IOC$MAPVBLK	CALL_MAPVBLK	No	No
IOC$MNTVER	CALL_MNTVER	No	No
IOC$MOVFRUSER	CALL_MOVFRUSER	No	No
IOC$MOVFRUSER2	CALL_MOVFRUSER2	No	No
IOC$MOVTOUSER	CALL_MOVTOUSER	No	No
IOC$MOVTOUSER2	CALL_MOVTOUSER2	No	No

[1]The JSB-based Step 1 routine is not supported by the OpenVMS AXP operating system Version 6.1.

(continued on next page)

Table 2–4 (Cont.) Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
IOC$PARSDEVNAM	CALL_PARSDEVNAM	No	No
IOC$POST_IRP	CALL_POST_IRP	No	No
IOC$PRIMITIVE_REQCHANH[1]	REQCHAN	No	No
IOC$PRIMITIVE_REQCHANL[1]	REQCHAN	No	No
IOC$PRIMITIVE_WFIKPCH	WFIKPCH	No	No
IOC$PRIMITIVE_WFIRLCH	WFIRLCH	No	No
IOC$PTETOPFN	CALL_PTETOPFN	No	R0 and R1
IOC$QNXTSEG1	CALL_QNXTSEG1	No	No
IOC$RELCHAN	RELCHAN	No	No
IOC$REQCOM	REQCOM	No	No
IOC$SEARCHDEV	CALL_SEARCHDEV	No	No
IOC$SEARCHINT	CALL_SEARCHINT	No	No
IOC$SEVER_UCB	CALL_SEVER_UCB	No	No
IOC$SIMREQCOM	CALL_SIMREQCOM	No	No
IOC$THREADCRB	CALL_THREADCRB	No	R0
MMG$IOLOCK	CALL_IOLOCK	No	No
MMG$UNLOCK	CALL_UNLOCK	No	No
MT$CHECK_ACCESS[1]	CALL_CHECK_ ACCESS	Yes	No
SCH$IOLOCKR	CALL_IOLOCKR	No	R1
SCH$IOLOCKW	CALL_IOLOCKW	No	No
SCH$IOUNLOCK	CALL_IOUNLOCK	No	No

[1]The JSB-based Step 1 routine is not supported by the OpenVMS AXP operating system Version 6.1.

2.8 Compiling, Linking, and Loading Step 2 Drivers

After you convert a Step 1 driver to a Step 2 driver, you must compile, link, and load it. For more information about compiling, linking, and loading Step 2 drivers, see the *OpenVMS AXP Device Support: Developer's Guide.*

3
Handling More Complex Situations

This chapter describes the Step 2 conversion situations that might be too unusual or too complex for the guidelines in Chapter 2.

3.1 Composite FDT Routines

A composite FDT routine is required when a single I/O function code must be processed by more than one upper-level FDT routine. Step 2 FDT dispatching only provides for a single upper-level routine for each I/O function code. When this is not sufficient, the general solution is to write a new upper-level FDT routine that sequentially calls each of the required upper-level FDT routines (checking status on return from each call). Another possible solution is to call the required second upper-level FDT routine at the appropriate point in the first upper-level FDT routine. The need for a composite FDT routine is automatically detected at compile time.

The following example shows a Step 1 FDT declaration.

```
FUNCTAB MY_FDT_ACPCONTROL,-
                <ACPCONTROL>
FUNCTAB ACP$MODIFY,-
                 <ACPCONTROL,MODIFY>
```

Using the guidelines in Section 2.5, you can obtain the following Step 2 declaration:

```
FDT_ACT MY_FDT_ACPCONTROL,-
                <ACPCONTROL>
FDT_ACT ACP_STD$MODIFY,-
                <ACPCONTROL,MODIFY>
```

However, you will receive the following error message when you attempt to compile the driver:

```
%AMAC-E-GENERROR, generated ERROR: 0 Multiple actions defined for function IO$_ACPCONTROL
```

To correct the source of the error, you must do the following:

1. Write a new upper-level FDT routine. This routine is a composite FDT routine that should call all the upper-level FDT routines listed by the FDT_ ACT macros for the function that has multiple actions. For example, you would write a routine like the following:

```
MY_FDT_ACPCONTROL_COMP:
       $DRIVER_FDT_ENTRY
                              ;  First FDT routine for IO$_ACPCONTROL
            PUSHL   R6        ;    P4 = CCB
            PUSHL   R5        ;    P3 = UCB
            PUSHL   R4        ;    P2 = PCB
            PUSHL   R3        ;    P1 = IRP
            CALLS   #4,MY_FDT_ACPCONTROL
            BLBC    R0,900$   ;    Quit if done
                              ;  Second FDT routine for IO$_ACPCONTROL
            CALL_ACP_MODIFY
900$:       RET               ;   Return status
```

2. Examine any of your driver-supplied upper-level FDT routines that you call from a composite FDT routine. With the exception of the last routine called in the composite routine, all the others will have at least one RSB exit path in their Step 1 version. (See Section 2.5.5.) You must convert this RSB as follows:

   ```
   MOVL #SS$_NORMAL,R0
   RET
   ```

   In a Step 1 driver, the RSB would have returned control to the FDT dispatching loop, so that the next upper-level FDT routine could be invoked. In a Step 2 driver, you must return a successful status, so that your composite FDT routine continues. Remember that the SS$_FDT_COMPL warning status will be returned by an upper-level FDT routine if FDT processing has completed and should not be continued.

3. Remove the function with multiple actions from all FDT_ACT macros. Then add a new FDT_ACT macro that invokes the new composite FDT routine for the function. In this example, you would write:

   ```
   FDT_ACT MY_FDT_ACPCONTROL_COMP, <ACPCONTROL>

   FDT_ACT ACP_STD$MODIFY, <MODIFY>
   ```

In many cases, a simpler solution is also possible. If you have a function that has multiple actions defined by FDT_ACT macros and the first FDT_ACT macro that references that function does not also include other functions, then you could convert your existing upper-level FDT routine into a composite FDT routine. You can do this by inserting the calls for the remaining upper-level FDT routines at the point where the first upper-level FDT routine would have returned to the Step 1 FDT dispatcher via an RSB instruction. This is the case in the previous example. Thus, if the Step 1 version of MY_FDT_ACPCONTROL looks like the following:

```
MY_FDT_ACPCONTROL:
     .JSB_ENTRY
     ...            ;driver-specific processing
     RSB            ;return to FDT dispatcher
```

Then the Step 2 composite version would look like the following:

```
MY_FDT_ACPCONTROL:
   $DRIVER_FDT_ENTRY
   ...              ;driver-specific processing
   CALL_ACP_MODIFY
   RET
```

3.2 Error Routine Callback Changes

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:

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

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

   If the error callback routine alters any nonscratch register as defined by the calling standard, you must add it to the preserve list. You can do this by using the **.SET_REGISTERS** directive or the **preserve** parameter on the $DRIVER_ERRRTN_ENTRY macro. For example, many error routines call EXE$DEANONPAGED or EXE$DEANONPGDSIZ, which destroy the contents of R2. You should specify **.SET_REGISTERS WRITTEN=<R2>**.

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

3. Replace the JSB to EXE$READLOCK_ERR, EXE$MODIFYLOCK_ERR, or EXE$WRITELOCK_ERR with the corresponding JSB-replacement macros: CALL_READLOCK_ERR, CALL_MODIFYLOCK_ERR, or CALL_ WRITELOCK_ERR.

   For more information, see *OpenVMS AXP Device Support: Reference*.

3.3 Converting Driver-Supplied FDT Support Routines to Call Interfaces

To convert driver-supplied FDT support routines to call interfaces, follow the procedure described in this section. Note that although this method is more efficient than the one described in Chapter 2, it requires that you make more changes to your source code.

1. Decide what the calling convention is for each of your FDT support routines.
2. Replace .JSB_ENTRY with .CALL_ENTRY at support routine entry points.
3. Within your converted support routines, you must refer to the routine parameters using the appropriate AP offsets. One way to do this is to copy the standard parameters into the registers used by the JSB interface.
4. Make sure that all driver-supplied FDT routines return status in R0.
5. All places that invoke your support routines via a JSB instruction must be changed to invoke the modified support routine via a CALLS instruction after having pushed the actual parameter values.
6. After each of these calls, you must also check the return status. For non-success status values (particularly SS$_FDT_COMPL), you must return to your caller.

   Using .JSB_ENTRY and the FDT completion macros, it is possible to write an FDT support routine that does not return to its caller in the event of an error. Once you convert to standard call interfaces, however, the flow of control always returns to the caller of the support routine.

Note

If any informational messages like the following are displayed, you have probably missed a .JSB_ENTRY FDT support routine or a branch between some other .JSB_ENTRY routine and an FDT support routine.

```
%AMAC-I-RETINJSB, RET in JSB_ENTRY
```

Once you have converted all your FDT support routines to standard call interfaces, you can eliminate many of the registers saves and restores that are generated by the default register preserve list on the $DRIVER_FDT_ENTRY macro. The default preserve list on the $DRIVER_FDT_ENTRY macro saves every nonscratch register to protect against a potential RET-under-JSB inside a .JSB_ENTRY FDT support routine. At the very least, you should be able to reduce the preserve list to **PRESERVE=<R2,R9,R10,R11>** to cover the registers that were allowed to be scratched by Step 1 upper-level FDT routines. You can reduce this list further, if you know that your FDT routine is not altering these registers, or if you rely on the .SET_REGISTERS directive and the register autopreserve feature of the MACRO-32 compiler,

3.4 Converting the Start I/O Code Path to Call Interfaces

Fork, special kernel AST, system timer expiration, and device interrupt timeout routines that are called by the OpenVMS exec can use either a standard call or the traditional JSB interface described in Chapter 2.

To convert the Start I/O Code Path to call standard interfaces in drivers written in MACRO-32, follow the procedure in Section 3.4.1. For a quick summary of the differences between using ENVIRONMENT=CALL and ENVIRONMENT=JSB, see Section 3.4.2. A detailed description of the Start I/O to REQCOM conversion implications for Step 1 drivers is available in *OpenVMS AXP Device Support: Reference*.

3.4.1 Start I/O Call Interface Conversion Procedure

To convert the Start I/O Code Path to call standard interfaces in drivers written in MACRO-32, follow these steps:

1. 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.
2. Use the DDTAB macro keywords

   altstart instead of **jsb_altstart**
   start instead of **jsb_start**
3. Use the ENVIRONMENT=CALL keyword parameter on the FORK, FORK_ROUTINE, FORK_WAIT, IOFORK, REQCOM, REQCHAN, REQPCHAN, WFIKPCH, and WFIRLCH macros.
4. 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.
5. Use the RET instruction instead of the RSB instruction to return from all of the previous standard call interface routines.

6. Use the scratch registers as defined by 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.

7. 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 code sequence. Note that although the following code appears more complex, it results in code that is just as efficient as that produced by the preceding example.

```
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
```

   For more details about this code sequence, see the description of the FORK ROUTINE interface in *OpenVMS AXP Device Support: Reference*.

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

8. Examine the interroutine branches between the previous routines and other routines in the same modules and change these routines to standard call interfaces.

9. If you encounter any of the following MACRO-32 compiler diagnostic messages, examine the relevant source:

```
%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 you can receive the last three diagnostic messages under acceptable circumstances. If this happens, you should document the reasons and consider disabling the diagnostic message by bracketing the smallest possible section of relevant code as follows:

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

In particular, the use of a RET from a JSB entry routine may be allowable in a Step 2 driver in the context of complex FDT routines. (For more information, see Section 2.5.4.) However, if you change the source code to avoid the need for a RET in a JSB routine, you can improve the performance of the code path. (For more information, see Section 3.3.)

3.4.2 Simple Fork Macro Differences

This section summarizes the differences between using the ENVIRONMENT=CALL and ENVIRONMENT=JSB parameters on the following simple fork macros:

FORK
FORK_ROUTINE
FORK_WAIT
IOFORK
REQCHAN
REQPCHAN
REQCOM
WFIKPCH
WFIRLCH

For more information about the parameters on these macros, see *OpenVMS AXP Device Support: Reference.*

3.4.2.1 Fork Routine End Instruction

Some simple fork macros generate an instruction that ends the current routine and returns control to the routine's caller. In a .JSB_ENTRY routine the appropriate end instruction is an RSB. However, a .CALL_ENTRY routine requires a RET instruction. Table 3–1 lists the simple fork macros whose fork routine end instruction is determined by the ENVIRONMENT parameter.

Table 3–1 Fork Routine End Instruction

Macros	ENVIRONMENT=CALL	ENVIRONMENT=JSB
FORK[1]	RET	RSB
FORK_WAIT[1]	RET	RSB
IOFORK[1]	RET	RSB
REQCHAN	RET	RSB
REQPCHAN	RET	RSB
REQCOM	RET	RSB
WFIKPCH	RET	RSB
WFIRLCH	RET	RSB

[1] If you use the CONTINUE parameter, this macro does not generate a fork routine end instruction.

3.4.2.2 Scratch Registers

Using the ENVIRONMENT=CALL parameter affects the list of scratch registers on some simple fork macros. Table 3–2 summarizes the differences in scratch register usage that are visible to the caller's fork thread. All other implicit register inputs and outputs on the simple fork macros are the same.

Table 3–2 Registers Scratched in Caller's Fork Thread

Macros	ENVIRONMENT=CALL	ENVIRONMENT=JSB
FORK	R0,R1 scratched	R0,R1 preserved
	R3,R4 preserved	R3,R4 sratched
FORK_WAIT	R0,R1 scratched	R0,R1 preserved
IOFORK	R0,R1 scratched	R0,R1 preserved
	R3,R4 preserved	R3,R4 scratched

The following example, which is a Step 1 unit initialization routine, illustrates how dependence on scratch register usage can be hidden in existing code:

```
MY_UNIT_INIT:
     .JSB_ENTRY  INPUT=<R0,R4,R5>,OUTPUT=<R0>
     ...         ;code that doesn't alter R0
     FORK  ROUTINE=MY_UNIT_INIT_FORK
```

This routine does some work and then queues the routine MY_UNIT_INIT_FORK as a fork routine. A unit initialization routine must return a successful status back to its caller. The preceding sample routine does this as follows:

- R0 is set to SS$_NORMAL before entry into the Step 1 unit initialization routine.
- The FORK macro with the default ENVIRONMENT=JSB setting does not alter R0.
- The FORK macro generates an RSB instruction.

The Step 2 equivalent of this unit initialization routine uses a standard call interface and must use the ENVIRONMENT=CALL parameter on the FORK macro. However, in doing so, the SS$_NORMAL value held in R0 is destroyed. The following example shows how to avoid this problem:

```
MY_UNIT_INIT:
     $DRIVER_UNITINIT_ENTRY
     ...
     FORK   ROUTINE=MY_UNIT_INIT_FORK,-
            ENVIRONMENT=CALL,-
            CONTINUE=10$
 10$: MOVZWL #SS$_NORMAL,R0
     RET
```

3.4.2.3 Fork Routine Entry Point

Some simple fork macros generate a fork routine entry point. The type of entry point generated depends on which ENVIRONMENT parameter you use. The parameters to a traditional JSB interface fork routine are contained in registers R3, R4, and R5. In contrast, the parameters to a standard call fork routine are passed using the standard argument passing mechanism and are referenced using AP offsets. The following macros generate code that copies the standard arguments into registers R3, R4, and R5; thereby, facilitating the conversion of existing JSB interface fork routines to the standard call interface:

FORK
FORK_ROUTINE
FORK_WAIT
IOFORK
REQCHAN
REQPCHAN
WFIKPCH
WFIRLCH

Table 3–3 summarizes the differences in the fork routine entry points generated by the FORK, FORK_ROUTINE, FORK_WAIT, IO_FORK, REQCHAN, REQPCHAN, WFIKPCH, and WFIRLCH macros as determined by the ENVIRONMENT parameter. Note that the FORK, FORK_WAIT, and IOFORK macros do not generate a fork routine entry point if you use the ROUTINE parameter.

Table 3–3 Fork Routine Entry Points

Entry Point Attributes	ENVIRONMENT=CALL	ENVIRONMENT=JSB
Entry directive	.CALL_ENTRY	.JSB_ENTRY
Parameters	Accessed using AP offsets[1]	R3,R4,R5
Parameter fetch	Parameters copied to R3,R4,R5[2]	None
Allowable scratch registers	R0,R1	R0-R4

[1]The symbolic names for the AP offsets are FORKARG$_FR3, FORKARG$_FR4, and FORKARG$_FKB.

[2]The parameter copy can be disabled on the FORK_ROUTINE macro if the FETCH=NO parameter is specified.

3.5 Device Interrupt Timeouts

Device interrupt timeouts are handled differently for Step 2 drivers. For Step 1 drivers the UCB$L_FPC cell in the device unit control block (UCB) contained the procedure value of the routine that served as both the resume from interrupt routine and the interrupt timeout routine. These two routines are now separate. The new UCB cell UCB$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 now needs 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 *OpenVMS AXP Device Support: Reference*.

3.6 Obsolete Data Structure Cells

Some DDT and DPT data structure fields that supported Step 1 device drivers have been removed. Table 3–4 lists the obsolete Step 1 fields and the Step 2 fields that have similar functions.

Note that the Step 2 cells use different names because they point to routines whose interfaces are different or they point to data structures whose layout is significantly altered. For this reason, do not replace each reference to an obsolete Step 1 field with its corresponding Step 2 field without considering the routine interface and data structure changes.

Table 3–4 Obsolete Data Structure Cells

Obsolete Step 1 Field	Similar Step 2 Field
DDT$L_ALTSTART	DDT$PS_ALTSTART_2 or DDT$PS_ALTSTART_JSB
DDT$PS_ALTSTART	DDT$PS_ALTSTART_2 or DDT$PS_ALTSTART_JSB
DDT$L_CANCEL	DDT$PS_CANCEL_2
DDT$PS_CANCEL	DDT$PS_CANCEL_2
DDT$L_CANCEL_SELECTIVE	DDT$PS_CANCEL_SELECTIVE_2
DDT$PS_CANCEL_SELECTIVE	DDT$PS_CANCEL_SELECTIVE_2
DDT$L_CHANNEL_ASSIGN	DDT$PS_CHANNEL_ASSIGN_2
DDT$PS_CHANNEL_ASSIGN	DDT$PS_CHANNEL_ASSIGN_2
DDT$L_CLONEDUCB	DDT$PS_CLONEDUCB_2
DDT$PS_CLONEDUCB	DDT$PS_CLONEDUCB_2
DDT$L_CTRLINIT	DDT$PS_CTRLINIT_2
DDT$PS_CTRLINIT	DDT$PS_CTRLINIT_2
DDT$L_FDT	DDT$PS_FDT_2
DDT$PS_FDT	DDT$PS_FDT_2
DDT$L_MNTVER	DDT$PS_MNTVER_2
DDT$PS_MNTVER	DDT$PS_MNTVER_2
DDT$L_REGDUMP	DDT$PS_REGDUMP_2
DDT$PS_REGDUMP	DDT$PS_REGDUMP_2
DDT$L_START	DDT$PS_START_2 or DDT$PS_START_JSB
DDT$PS_START	DDT$PS_START_2 or DDT$PS_START_JSB
DDT$L_UNITINIT	DDT$PS_UNITINIT_2
DDT$PS_UNITINIT	DDT$PS_UNITINIT_2
DPT$PS_DELIVER	DPT$PS_DELIVER_2

3.7 Optimizing Step 2 Drivers

When you have successfully converted a Step 1 device driver to a Step 2 device driver, you can optimize the driver's performance by performing the tasks covered in Section 3.7.1 through Section 3.7.3.

3.7.1 Using JSB-Replacement Macros

You can replace a JSB to a system routine in a Step 1 driver with a macro. The JSB-replacement macro uses the same input registers and modifies 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.

As mentioned in Chapter 2, you should use the JSB-replacement macros in Table 2–4 instead of an explicit JSB to the listed JSB-interface system routines. A JSB-replacement macro is provided if the JSB-interface routine is no longer available or if the JSB-interface routine is less efficient than the new standard call version of the routine. The JSB-replacement macros use the register inputs and outputs that your existing Step 1 code expects. However, these macros directly invoke the new Step 2 standard call interface routines.

3.7.2 Avoid Fetching Unused Parameters

You can adapt a driver's use of the driver entry point macros, so that it more closely resembles 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 of 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 code 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.

3.7.3 Minimizing Register Preserve Lists

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 scratch 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 follow this recommendation only if the FDT processing initiated by the upper-level FDT action routine avoids the situation in which a subroutine call initiated by a JSB instruction is concluded by a RET instruction instead of an RSB. 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.

In some cases you may be able to remove all registers from the preserve list. Note that you can select an empty register preserve list for the driver entry point macros only by specifying **PRESERVE**=**NULL**. In contrast, if you specify **PRESERVE**=**<>**, you will get the default value for the register preserve list and not an empty preserve list.

Index

D

E

F

I

J

L

M

P

S

U