EDUCATIONAL
SERVICES

VAX/VMS
Training

VAX/VMS
Device Driver
FDT Routines

digital

EY-2278E-MF-0001

FDT ROUTINES

Prepared by Educational Services
of
Digital Equipment Corporation

First Edition, October 1984

Printed in U.S.A.

The manuscript for this book was created using DIGITAL Standard Runoff. Book production was done by Educational Services Development and Publishing in Nashua, NH.

CONTENTS

FIGURES

INTRODUCTION

FDT routines are called from EXE$QIO to validate the P1-P6 arguments in a $QIO request. VMS contains many device-independent FDT routines that drivers may call. When a system routine does not exist to perform the desired validation, user-written FDT routines may be used. System FDT routines should be called when possible to minimize driver size, and to reduce the possibility of an error. FDT routines should be regarded, logically, as extensions to EXE$QIO.

FDT routines are different for drivers performing buffered and direct I/O, although there are some common elements. This module discusses writing FDT routines for devices performing buffered or direct I/O.

OBJECTIVES

Upon completion of this module, you will be able to:

1. Define position-independent and reentrant code.
2. Write FDT routines for drivers performing buffered or direct I/O.
3. List the exit methods used in FDT routines (and state where each method is appropriate).

RESOURCE

1. Guide to Writing a Device Driver for VAX/VMS

LEARNING ACTIVITIES

1. Study FDT routines in existing drivers.

2. Study system FDT routines (either source code in module SYSQIOREQ, or descriptions in Appendix C of the driver manual).

TOPICS

- FDT Overview
- Buffered I/O vs. Direct I/O
- FDT Routines for Buffered I/O
- FDT Routines for Direct I/O
- Exit Methods from FDT Routines

POSITION-INDEPENDENT CODE (PIC)

Since the driver is loaded into nonpaged system memory, and the starting address in nonpaged pool is unknown (it varies each time the driver is loaded), the driver must consist of position-independent code only. Position-independent code works the same regardless of its position in memory. To make driver code PIC:

- Reference I/O data structures by loading the address of a data structure into a general register, and use displacement addressing to reference entries in the data structure.

- Transfer control only to relative addresses within the driver, and to global addresses in the system symbol table (SYS.STB).

- Precede system addresses with G^.

- Use MOVA whenever an address is moved.

REENTRANT CODE

A device driver is called repeatedly to process I/O requests and interrupts. Only one copy of the driver exists in nonpaged memory for all devices being serviced by the driver. Since the driver may not complete one I/O operation before another is started, the code must be reentrant. Reentrant code works the same every time (same logic, same output) given the same input. In contrast, self-modifying code is not reentrant. To make code reentrant:

- Do not store temporary data in local buffers. All temporary storage must be contained within the UCB.

FDT ROUTINE OVERVIEW

FDT Routines

- are invoked by EXE$QIO.
- run at IPL=2 (in kernel mode, on kernel stack).
- run in process context.
- may modify RØ-R2, and R9-R11.
- must save and restore any other register(s) used.

When called, registers R3-R8, and AP, contain:

```
R3=  IRP address
R4=  current process PCB address
R5=  UCB address
R6=  CCB address
R7=  I/O function code in $QIO call
R8=  address of FDT routine about to be called
AP=  address of P1 parameter (P2-P6 follow P1)
     (AP modified to this by $QIO)
```

FDT routines are responsible for

- validating P1-P6, and storing them in IRP.
- moving data from user buffer(s) to system buffer(s).
- locking pages in memory.

Restrictions on FDT Routines

- Must save and restore any registers altered except RØ-R2, and R9-R11.

- Must not lower IPL below 2 (IPL$_ASTDEL).

- If IPL is raised, it must be lowered to 2 before exiting.

- Must restore stack to its original state before exiting.

- Normally, should only alter fields in IRP. If UCB field needs to be read/written, should queue IRP to driver start I/O routine to synchronize with outstanding I/O requests. If not queuing request to driver, should raise (and later lower) IPL to device fork level.

- Cannot call on system services, since they often lower IPL to Ø.

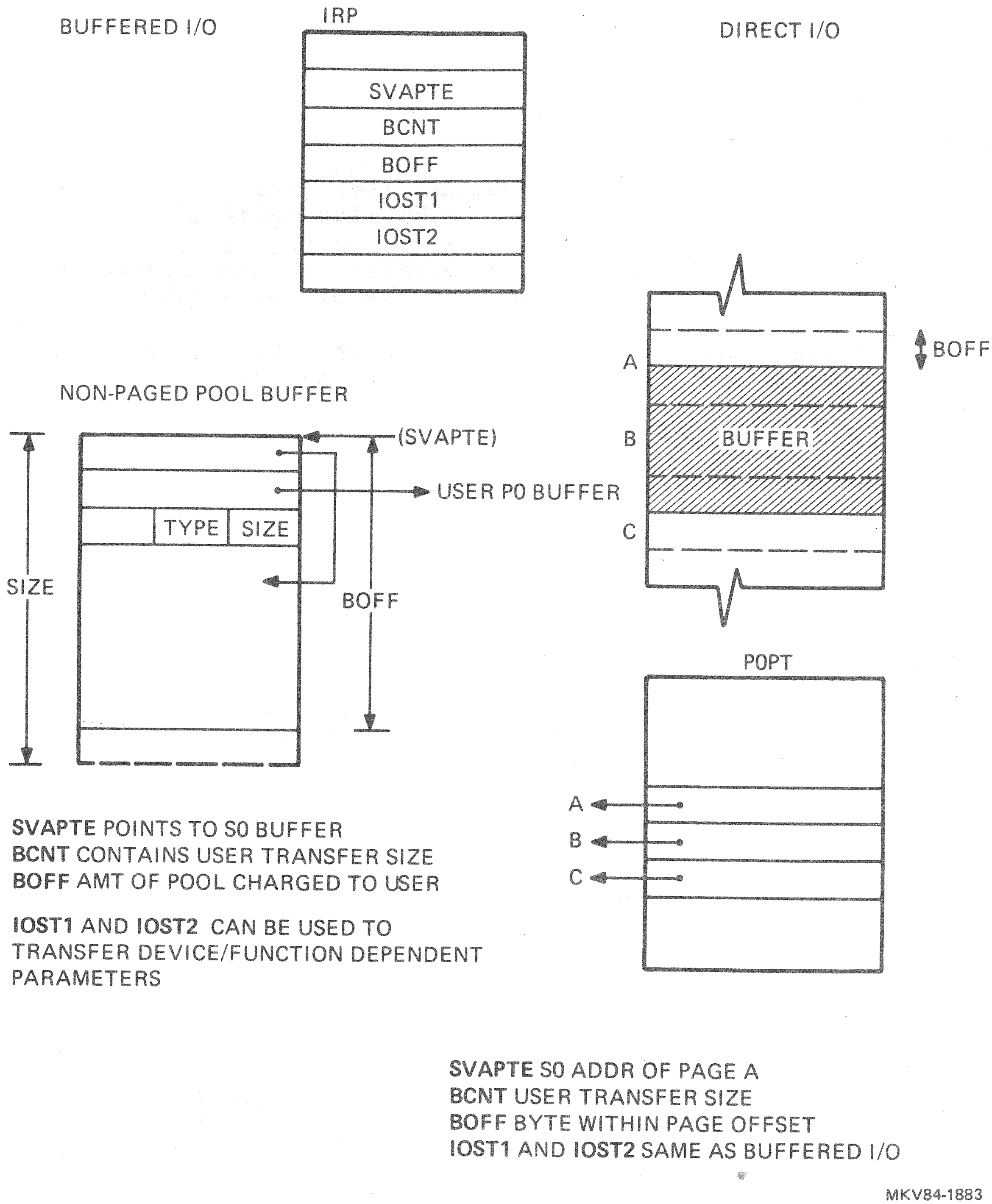

Figure 6-1 Use of IRP vs. Buffered/Direct I/O

FDT Routines for Buffered I/O

The following steps need to be taken by any FDT routine performing buffered I/O:

1. Check accessibility of user buffer(s) [normally P1]. This is done by a:

   ```
       JSB G^EXE$READCHK (for READ QIO)
   or  JSB G^EXE$WRITECHK (for WRITE QIO)
   ```

   The inputs/outputs/side effects of the system routines are listed in Appendix C of the Driver Manual.

2. Check buffer quota. Use buffer size passed by user, [normally P2] + 12 bytes [system overhead].

   ```
   JSB G^EXE$BUFFRQUOTA
   ```

3. Allocate buffer from nonpaged pool (get address in R2)

   ```
   JSB G^EXE$ALLOCBUF
   ```

   (Careful! This routine destroys R3; make sure you save and restore it with a PUSHR #^M<R3>, POPR #^M<R3> sequence.)

   Note that the size of the buffer you allocate should be the size the user requested + 12 bytes for system overhead.

4. Initialize the first two longwords of the buffer as shown in Figure 6-2 (system fills in the third longword when buffer is allocated):

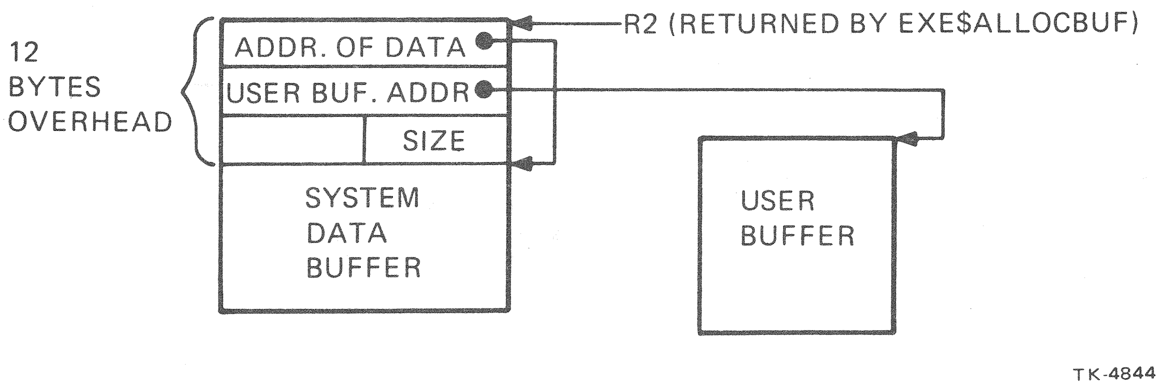

Figure 6-2 Format of Buffered I/O Buffer

5. Charge process for the buffer in the following manner:

> NOTE
>
> IRP$W_BOFF(R3) should contain the number of bytes charged from the process's quota. I/O post will add what is in this field to the process's quota when the I/O completes (for buffered I/O).

(assume RØ is a scratch register;

R1 contains the buffer size, including the twelve bytes of overhead;

R4 contains the PCB address)

```
MOVL PCB$L_JIB(R4),RØ      ; get JIB address

SUBL R1,JIB$L_BYTCNT(RØ)   ; charge process for buffer
```

Make sure you include the $PCBDEF and $JIBDEF macros in your driver to define the PCB$ and JIB$ offsets.

6. For a WRITE operation, transfer data from the user buffer to the system buffer with a MOVC instruction. Again, be careful, since MOVC instructions destroy RØ-R5, and registers R3-R5, at least, should be saved and restored before exiting.

7. Make sure any of the P1-P6 parameters required by the start I/O routines are placed into fields in the IRP. Some of the fields are copied by system routines (such as EXE$READCHK or EXE$WRITECHK). Consult Appendix C of the driver manual or the source code for specifics; normally,

```
P1 --> Second longword of nonpaged pool buffer
P2 --> IRP$W_BCNT(R3)
P3 --> *
P4 --> IRP$B_CARCON(R3)
P5 --> *
P6 --> *
```

* Up to you; often-used fields include IRP$L_MEDIA and IRP$L_MEDIA+4.

If you need to pass more information to the start I/O routine than can be placed in the IRP (for either buffered or direct I/O), you can allocate and use one or more I/O Request Packet Extensions (IRPEs). The following instructions illustrate how to allocate and initialize an IRPE:

```
PUSHL      R3                          ; save IRP address
JSB        G^EXE$ALLOCIRP              ; allocate IRPE (addr in R2)
POPL       R3                          ; restore IRP address
MOVB       #DYN$C_IRPE, IRPE$B_TYPE(R2)  ; change type
                                       ; to IRPE (not IRP)
CLRW       IRPE$W_STS(R2)              ; clear status bit in IRPE
MOVL       R2, IRP$L_EXTEND(R3)        ; link IRPE to IRP
BISW       #IRP$M_EXTEND, IRP$W_STS(R3)  ; set EXTEND bit
fill in other IRPE                     ; indicating IRPE present
fields (using IRPE$...
offsets to reference the fields)
```

You can link any number of IRPEs by allocating the number you need, setting the IRP$M_EXTEND bit in each IRPE's status word (IRP$W_STS) except the last, and linking the IRPEs via the IRPE$L_EXTEND fields. I/O Post automatically deallocates all linked IRPEs when the I/O request completes. Be careful to clear the status word of the last IRPE allocated (since the IRPE allocated may or may not have that word clear), since I/O Post will test the EXTEND bit in the status word, and, if set, use whatever it finds in IRPE$L_EXTEND as a pointer to another IRPE. (Include $IRPEDEF in your driver to define the IRPE$ offsets.)

Also, if you are doing direct I/O, make sure the IRPE$L_SVAPTE1 and IRPE$L_SVAPTE2 fields are set to Ø. If these fields are not zero, IOPOST will interpret the values as the addresses of pages to unlock (with IRPE$W_BOFF1, IRPE$L_BCNT1, IRPE$W_BOFF2, and IRPE$L_BCNT2 giving the byte offset in page, and size, of the locked buffers).

FDT Routines for Direct I/O

Any FDT routine performing direct I/O must:

1. Check the accessibility of user buffer as in buffered I/O.

2. Lock buffer pages in memory by calling routine MMG$IOLOCK (in module IOLOCK of the executive). [JSB G^MMG$IOLOCK]

3. Transfer any device-specific information into the IRP as in buffered I/O.

Normally, drivers performing direct I/O call on EXE$READ or EXE$WRITE to perform the above steps. (See descriptions in Chapter 8 of the Driver Manual.)

The routines set:

	IRP$L_SVAPTE(R3)	Address of PTE mapping the starting address of buffer
P2 -->	IRP$W_BCNT(R3)	Byte count for transfer
	IRP$W_BOFF(R3)	Starting byte in first page of transfer
P4 -->	IRP$B_CARCON(R3)	For whatever purpose you would like

Exit Methods from FDT

Purpose	Method	Comment(s)
Return to FDT dispatcher ($QIO)	RSB	A future FDT will not RSB back to $QIO
Abort the Request	JMP G^EXE$ABORTIO	Device-independent error in sufficient buffer quota; NOIOSB written
I/O operation finished	JMP G^EXE$FINISHIO	RØ + R1 placed in IOSB RØ = SS$_NORMAL R1 = dev_dependent info
	JMP G^EXE$FINISHIOC	Same as EXE$FINISHIO except R1 set to Ø before IOSB written
Queue IRP to UCB	JMP G^EXE$QIODRVPKT	If UCB busy, IRP queued; if UCB not busy, I/O started
Queue IRP to ACP	JMP G^EXE$QIOACPPKT	Place IRP on ACP queue (see Related Topics module)
Queue IRP to XQP	JMP G^EXE$QXQPPKT	Place IRP on XQP queue (see Related Topics module)
Queue IRP to alternate entry	JMP G^EXE$ALTQUEPKT	Control transferred to driver's alternate start I/O point entry point

SYSTEM-SUPPLIED FDT ROUTINES

Routine	Purpose
	Calls EXE$QIODRVPKT
EXE$ONEPARM	- plus copies P1 to IRP$L_MEDIA
EXE$ZEROPARM	- plus clears IRP$L_MEDIA
EXE$SETMODE	- plus moves device characteristics (quadword) to IRP$L_MEDIA and IRP$L_MEDIA+4
	Calls EXE$FINISHIO
EXE$SENSEMODE	- plus copies UCB$L_DEVDEPEND to R1
EXE$SETCHAR	- plus writes UCB fields DEVCLASS, DEVTYPE, DEVBUFSIZ, and DEVDEPEND from I/O request
	Validates and Readies:
EXE$READ	- user buffer for DMA read
EXE$WRITE	- user buffer for DMA write
EXE$MODIFY	- user buffer for DMA read and write (modify)
	For source code see SYSQIOFDT.MAR

EY-2278E-MF-0001
Printed in U.S.A.