EDUCATIONAL
SERVICES

VAX/VMS
Training

VAX/VMS
Device Driver
Required
Driver Routines

digital

EY-2278E-MH-0001

REQUIRED DRIVER ROUTINES

Prepared by Educational Services
of
Digital Equipment Corporation

First Edition, October 1984

Printed in U.S.A.

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 not be used or copied except in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

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

EXAMPLES

INTRODUCTION

Although all drivers are individual in some respects, they have similar layouts. The general structure of all drivers includes:

- Driver Tables, Symbol Definitions, Macros
- FDT Routines
- Start I/O Routine
- Timeout Routine
- Interrupt Service Routine
- Cancel I/O Routine
- Controller Initialization Routine
- Unit Initialization Routine
- Register Dump Routine (Error Logging)

This module examines the routines that are required in all drivers (start I/O and interrupt service routines). The remaining routines are discussed in a separate module. Although FDT routines exist in all drivers, FDT routines are considered extensions to the $QIO system service (not a part of the driver), and are discussed in a separate module.

OBJECTIVES

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

1. List restrictions on the use of the instruction set when accessing device registers.

2. Write the following driver routines for drivers performing either buffered or direct I/O:

   - start I/O routine
   - interrupt service routine

RESOURCE

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

LEARNING ACTIVITIES

1. Study the source listings for the system macros invoked (in module SYSQIOREQ AND IOSUBNPAG).

2. Study the start I/O, timeout, and interrupt service routines in existing drivers.

TOPICS

- Instruction set limitations
- Start I/O routine
  - Defining symbolic offsets
  - Define extensions to UCB
  - Define symbolic bit positions/masks
  - Buffered I/O
  - Direct I/O
- Interrupt Service routine
- MASSBUS routines

RESTRICTIONS ON DRIVER CODE AND USE OF THE INSTRUCTION SET

- Must be position-independent code
  - load data structure references into general registers
  - reference contents of data structures with displacement addressing
  - precede system addresses with G^
- Must be reentrant code
  - all temporary storage contained in UCB, none on the stack
- Must use noninterruptible instructions (i.e., 2 operand) when referencing I/O space (i.e., device registers)
- UNIBUS drivers use only byte or word references to device registers. Do not use BBSx, BBCx (longword implementation in microcode)
- Registers in MBA and MASSBUS devices are longwords
- If system routines called, be careful to know exactly what the routine does.

DEFINING SYMBOLIC OFFSETS

Drivers always reference data block entries by symbolic offsets. Before making any such references, the symbolic offsets must be defined by invoking the corresponding system macros. Each of the following macros defines the symbolic offsets for its corresponding data block:

In general, for any data structure with symbolic offsets, those offsets can be defined by including a $ (structure_name)DEF macro.

```
$CRBDEF                 ;DEFINE CRB OFFSETS
$DDBDEF                 ;DEFINE DDB OFFSETS
$DDTDEF                 ;DEFINE DDT OFFSETS
$DIBDEF                 ;BUFFER OFFSETS FOR INPUT
                        ;BUFFER TO SET MODE/CHAR
$EMBDEF                 ;DEFINE EMB OFFSETS
$IDBDEF                 ;DEFINE IDB OFFSETS
$IRPDEF                 ;DEFINE IRP OFFSETS
$IRPEDEF                ;DEFINE IRPE OFFSETS
$JIBDEF                 ;DEFINE JIB OFFSETS
$MBADEF                 ;DEFINE MBA REGISTER OFFSETS
$PCBDEF                 ;DEFINE PCB OFFSETS
$UCBDEF                 ;DEFINE UCB OFFSETS
$VCBDEF                 ;DEFINE VCB OFFSETS
$VECDEF                 ;DEFINE INTR DISPATCH VECTOR OFFSETS
$WCBDEF                 ;DEFINE WCB OFFSETS
```

There are also many symbolic values the drivers can use after invoking the correct macros.

```
$DCDEF                  ;DEFINE DEVICE CLASSES AND TYPES
$DEVDEF                 ;DEFINE DEVICE CHARACTERISTICS
$DYNDEF                 ;DEFINE DATA STRUCTURE TYPES
$IODEF                  ;DEFINE I/O FUNCTION CODES
$IPLDEF                 ;DEFINE IPL LEVELS
$PRDEF                  ;DEFINE PROCESSOR REGISTERS
$PRnnnDEF               ;DEFINE PROCESSOR-SPECIFIC REGISTERS
                         nnn = 780, 750, 730
$PRVDEF                 ;DEFINE PRIVILEGE BIT DEFINITIONS
$SSDEF                  ;DEFINE SYS STATUS VALUES
```

Processor register space provides access to many CPU control/status registers. Processor-specific values are in $PRnnnDEF - for example, the 780 Interval Count Register is symbolic value PR780$_ICR in $PRO780DEF.

DEFINING EXTENSIONS TO THE UCB

Three macros are used to define extensions to the UCB: $DEFINI, $DEF, and $DEFEND. These macros also produce symbols which are used to reference the extended fields.

```
        $DEFINI         UCB             ; start extensions

.=UCB$K_LENGTH                          ; position to end of
                                        ; standard UCB

$DEF    UCB$B_BYTE_ONE      .BLKB 1     ; extension 1
$DEF    UCB$B_BYTE_TWO      .BLKB 1     ; extension 2
$DEF    UCB$W_WORD          .BLKW 1     ; extension 3
$DEF    UCB$L_LONG          .BLKL 1     ; extension 4

$DEF    UCB$K_NEW_LENGTH                ; for use as
                                        ; UCB size in
                                        ; DPTAB macro

        $DEFEND     UCB         ; done with extensions
```

This example adds four fields on the end of the UCB, two bytes, a word, and a longword. The driver can reference these fields symbolically, and use them as needed. Make sure the fields you add are "naturally" aligned (do not cross longword boundaries).

There are two extensions to the UCB that are not part of the standard UCB: the error-logging and disk extensions. To use these extensions (see Appendix A of the driver manual), replace

```
                .=UCB$K_LENGTH

with            .=UCB$L_DPC+4               for error log extensions

or              .=UCB$K_LCL_Disk_length     for disk (and error log)
                                            extensions
```

Your extensions are then placed after the nonstandard system extensions for error logging/disks. Note that the symbolic offsets for the nonstandard extensions are included in the $UCBDEF macro.

DEFINING SYMBOLIC BIT FIELDS (_VIELD MACRO)

- General Form:

```
$DEFINI         structure.name

 _VIELD         name.prefix,starting.bit.position, <-

                <bit.name,width,key.letter>, -
                                .
                                .
                                .
                <bit.name,width,key.letter> -
                >

$DEFEND
```

- Generates Symbols of the Form:

name.prefix __V__ bit.name = bit number (for BBxy, etc.)

name.prefix __/__ bit.name = bit mask (for BISx,BICx, etc.)

name.prefix __S__ bit.name = size of bit field

- Rules:

1. If bit.name is left out, no symbols are generated.
2. "S" symbol is generated only if the width is not blank.
3. If width is left out, it default to 1 and no "S" symbol is generated.
4. If key.letter is "M", "V", or "S", then "M" and "V" symbols are generated.
5. If key.letter is blank, then only "V" (no "M") symbol is generated; see also (2).

- $VIELD Macro:

The $VIELD macro is syntactically equivalent to _VIELD, except that it produces symbols of the form:

name.prefix $V_bit.name

This macro is used by the system to avoid duplicate names.

```
0000      1        ..TITLE TEST VIELD MACRO
0000      2
0000      3        $DEFINI                          SYMBOLS
0000      4                                         GENERATED
0000      5        _VIELD     QQQ,4<-
0000      6                   <ABIT,1,M>,-          M     S     V
0000      7                   <2BITS,2,M>,-         M     S     V
0000      8                   <NOWIDTH,,M>,-        M           V
0000      9                   <2MORE,2,V>,-         M     S     V
0000     10                   <NOWIDTH1,,V>,-       M           V
0000     11                   <SKEY,2,S>,-          M     S     V
0000     12                   <SKEYNO,,S>,-         M           V
0000     13                   <NOKEY,>,-                        V
0000     14                   <NOKEYW,3,>,-               S     V
0000     15                   <,4,>,-
0000     16                   <,4,M>,-
0000     17                   <,4,S>,-
0000     18                   <ONEMORE,3,M>-        M     S     V
0000     19                   >
0000     20
0000     21        $DEFEND
0000     22
0000     23        .END
```

```
BIT...              =  00000021
GBL...              =  00000000
QQQ_M_2BITS         =  00000060
QQQ_M_2MORE         =  00000300
QQQ_M_ABIT          =  00000010
QQQ_M_NOWIDTH       =  00000080
QQQ_M_NOWIDTH1      =  00000400
QQQ_M_ONEMORE       =  C0000000
QQQ_M_SKEY          =  00001800
QQQ_M_SKEYNO        =  00002000
QQQ_S_2BITS         =  00000002
QQQ_S_2MORE         =  00000002
QQQ_S_ABIT          =  00000001
QQQ_S_NOKEYW        =  00000003
QQQ_S_ONEMORE       =  00000003
QQQ_S_SKEY          =  00000002
QQQ_V_2BITS         =  00000005
QQQ_V_2MORE         =  00000008
QQQ_V_ABIT          =  00000004
QQQ_V_NOKEY         =  0000000E
QQQ_V_NOKEYW        =  0000000F
QQQ_V_NOWIDTH       =  00000007
QQQ_V_NOWIDTH1      =  0000000A
QQQ_V_ONEMORE       =  0000000B
QQQ_V_SKEY          =  0000000B
QQQ_V_SKEYNO        =  0000000D
```

START I/O ROUTINE

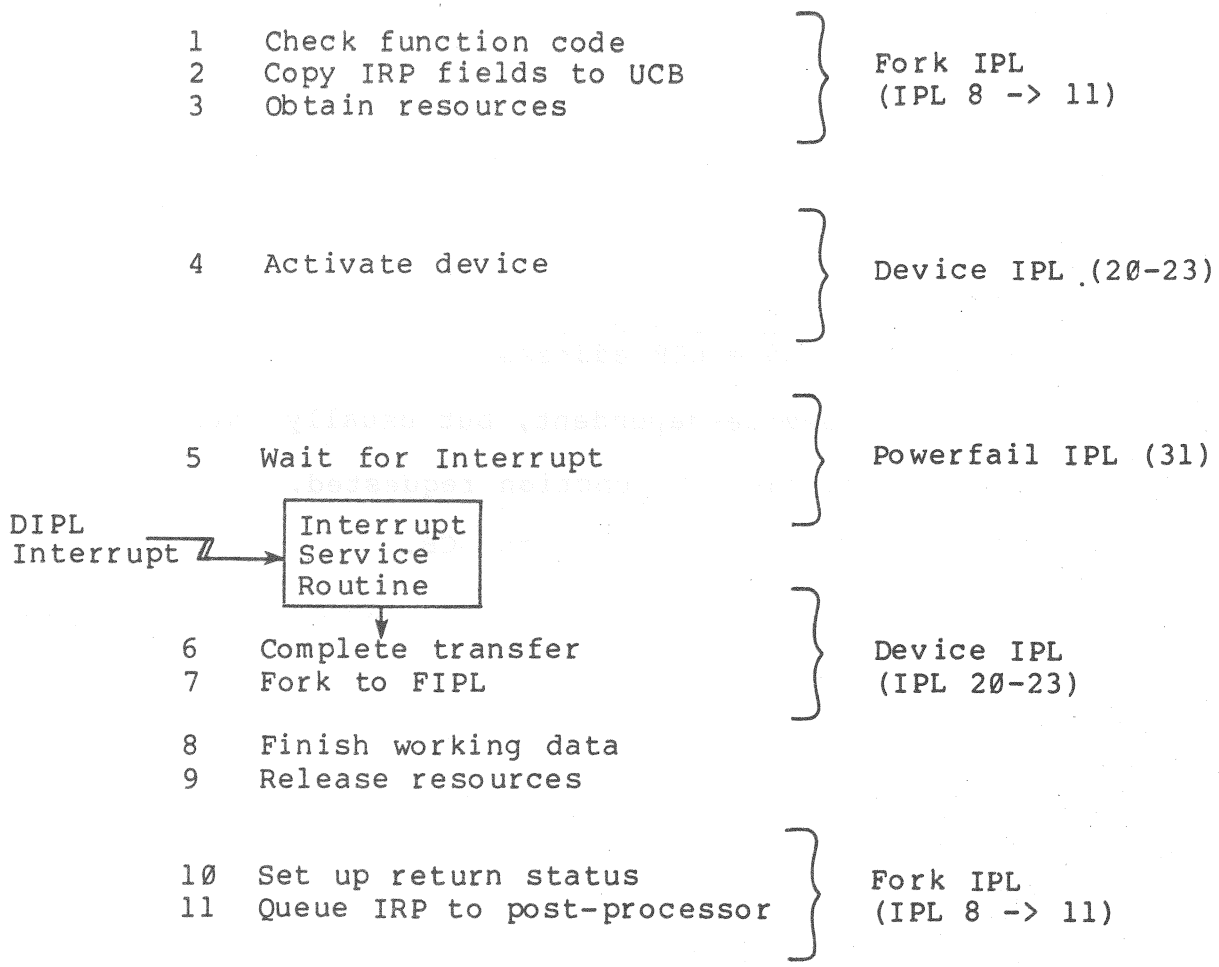
- The start I/O routine activates a device, then waits for an interrupt or timeout.
- The start I/O routine does **not** run in the context of a user process and can only reference SØ space.
- The start I/O routine runs in kernel mode, at IPL $\geq$ fork IPL. Hence, no page faults are allowed.
- When called, either from EXE$QIODRVPKT or IOC$REQCOM:

  R3 = IRP address
  R5 = UCB address

- Processing is device-dependent, but usually involves:
  - determining the I/O function requested.
  - copying fields from IRP into UCB.
  - obtaining and initializing resources (e.g., controller, data path, mapping registers).
  - modifying device registers to activate the device.
- If the alternate start I/O routine is called (from EXE$ALTQUEPKT):
  - no check is made to see if the device is busy (useful for full-duplex operation).
  - the SVAPTE, BCNT, and BOFF fields are not copied from IRP to UCB.
  - the cancel and timeout bits in the UCB are not cleared.
- The alternate start I/O routine is not a required driver routine.

Generic Start I/O

Start I/O:



Buffered I/O		Direct I/O
Check fxn code	1	Check fxn code
transfer IRP fields -> UCB	2	transfer IRP fields to UCB
REQPCHAN	3	REQPCHAN REQDPR REQMPR LOADUBA
DSBINT UCB$B_DIPL(RS) load device registers SETIPL #IPL$_POWER check for powerfail set device "go" bit	4	DSBINT UCB$B_DIPL(R5) load device registers SETIPL #IPL$_POWER check for powerfail set device "go" bit
WFIKPCH	5	WFIKPCH
(Device generates interrupt, code resumed by Interrupt Service routine)		
read device registers	6	read device registers
IO FORK	7	IOFORK
(Code resumed by fork dispatcher)		
handle data	8	handle data
RELCHAN	9	PURDPR RELDPR RELMPR RELCHAN
load final IOSB status into RØ, R1	1Ø	load final IOSB status into RØ R1
REQCOM	11	REQCOM

Special Considerations for Full-Duplex Drivers

- Must have more than one I/O request active at a time; therefore, some I/O requests may be queued to the driver's alternate start I/O routine by FDT routines. For example

  - all read requests terminate with EXE$QIODRVPKT.
  - all write requests terminate with EXE$ALTQUEPKT.

- Must have more than one interrupt expected at a time; therefore, need more than one fork block with saved return PC. For example:

  - can have extensions to UCB to contain fork block, and before forking (maybe with FORK macro instead of WFIKPCH), alter R5 to point to extended part of UCB (that has space for R3, R4, and PC).

  - can have listhead in UCB off which "outstanding" IRPs are queued, each of which contains a fork block (again, change R5 to point to the IRP before forking).

- IRPs queued from EXE$ALTQUEPKT must be passed to the I/O post-processing routines without issuing the REQCOM macro. That macro results in the next IRP queued to the UCB by EXE$QIODRVPKT being started, and the current IRP queued by EXE$QIODRVPKT may not yet have terminated. Therefore, routine COM$POST (in module COMDRVSUB) is usually called to place IRPs on the post-processing queue.

- The driver must manage and synchronize all I/O activity.

- The driver should operate almost exclusively at device IPL to block out device interrupts to synchronize with multiple I/O request processing.

- See the terminal driver and DMC-11 driver for examples of how full-duplex operation is implemented.

INTERRUPT SERVICE ROUTINE

Interrupt service routines for UNIBUS device drivers are invoked

- at device IPL.
- in response to a hardware interrupt.
- in kernel mode on interrupt stack.
- with the stack containing:

Ø(SP)	pointer to address of IDB
4(SP)-24(SP)	saved RØ-R5
28(SP)	PC at time of interrupt
32(SP)	PSL at time of interrupt

Interrupt service routines typically

- locate the UCB for the device (from the IDB OWNER or UCBLIST fields).
- determine whether the interrupt was expected (solicited) or unexpected (unsolicited).

  if unexpected, decide whether to reject or process the interrupt.

  if expected, restore the driver fork process context from the UCB.
- transfer control to the stored PC (via JSB).
- dismiss the interrupt by restoring RØ-R5 and issuing an REI instruction (will happen when control is returned following the IOFORK macro).

```
        ISR:

             MOVL @(SP)+,R3             ; get IDB address and
                                        ; remove it from stack
1            MOVQ IDB$L_CSR(R3),R4      ; get CSR and owner UCB
                                        ; addresses
2            BBCC #UCB$V_INT,-          ; see if interrupt
                  UCB$W_STS(R5),-       ; is expected
                  10$
3             MOVL UCB$L_FR3(R5),R3     ; restore fork R3
4             JSB  @UCB$L_FPC(R5)       ; resume fork process

5       10$: MOVQ (SP)+, RØ             ; dismiss the
             MOVQ (SP)+, R2             ; interrupt after
             MOVQ (SP)+, R4             ; restoring registers
             REI                        ; RØ-R5
```

Example 8-1 Sample Interrupt Service Routine

The following notes are keyed to Example 8-1, and indicate where more complex processing may be needed.

1. The MOVQ instruction places the CSR address in R4, and the owner UCB address in R5. If there is no owner (following a WFIRLCH macro), the interrupt service routine must determine which unit caused the interrupt. Sometimes, devices have an interrupt summary register which is read to determine the unit number generating the interrupt. In this case, the unit number is read, and used as an index into the UCBLIST table in the IDB to find the UCB address, which is placed in R5.

   If no device register is available, separate interrupt service routines may have to be written for each vector that an interrupt can occur at. The only difference between these routines would be the UCBLIST entry used for the UCB. The units would have to be connected (in SYSGEN) in a particular order, so that the appropriate interrupt service routine would be found. In any case, a separate entry must be made in the driver prologue table, which specifies the interrupt service routine to be used for each interrupt vector (several vectors may specify the same interrupt service routine).

2. If an unexpected interrupt has meaning for your device, you may want to process the interrupt (e.g., card reader turned on-line). You should still dismiss the interrupt by restoring RØ-R5 and issuing an REI.

3. R4 is not restored, since your driver needs the CSR address in R4. Normally, your driver would have the CSR address in R4 before it called WFIKPCH. If you need R4 restored to something other than the CSR, you could do it here.

4. A JSB is used to allow control to return at a later time, (following IOFORK). After the JSB, you should always restore RØ-R5, and issue an REI. Execution resumes at the instruction following the WFIKPCH or WFIRLCH macro.

5. Your routines must explicitly save and restore any registers except RØ-R5 which you want to use. Three MOVQs happen to be a little faster than a POPR instruction in this case.

Interrupt Service Routines for MASSBUS Devices

Interrupt service routines for MASSBUS device drivers are invoked only for multidevice controllers. You must specify the DPT$M_SUBCNTRL bit in the FLAGS field of the DPTAB macro. When invoked, the stack differs slightly from UNIBUS drivers, in that only R2-R5 are saved, and the PC and PSL are found in MBA$INT, the MASSBUS adapter interrupt service routine in module MBAINTDSP in the executive, not the PC and PSL at the time of the interrupt.

The interrupt service routines typically perform the same functions UNIBUS interrupt service routines perform. In addition, they must clear the MBA attention summary bit for the interrupting device. They restore R2-R5 following the JSB call to the stored PC in the UCB followed by an REI. For example:

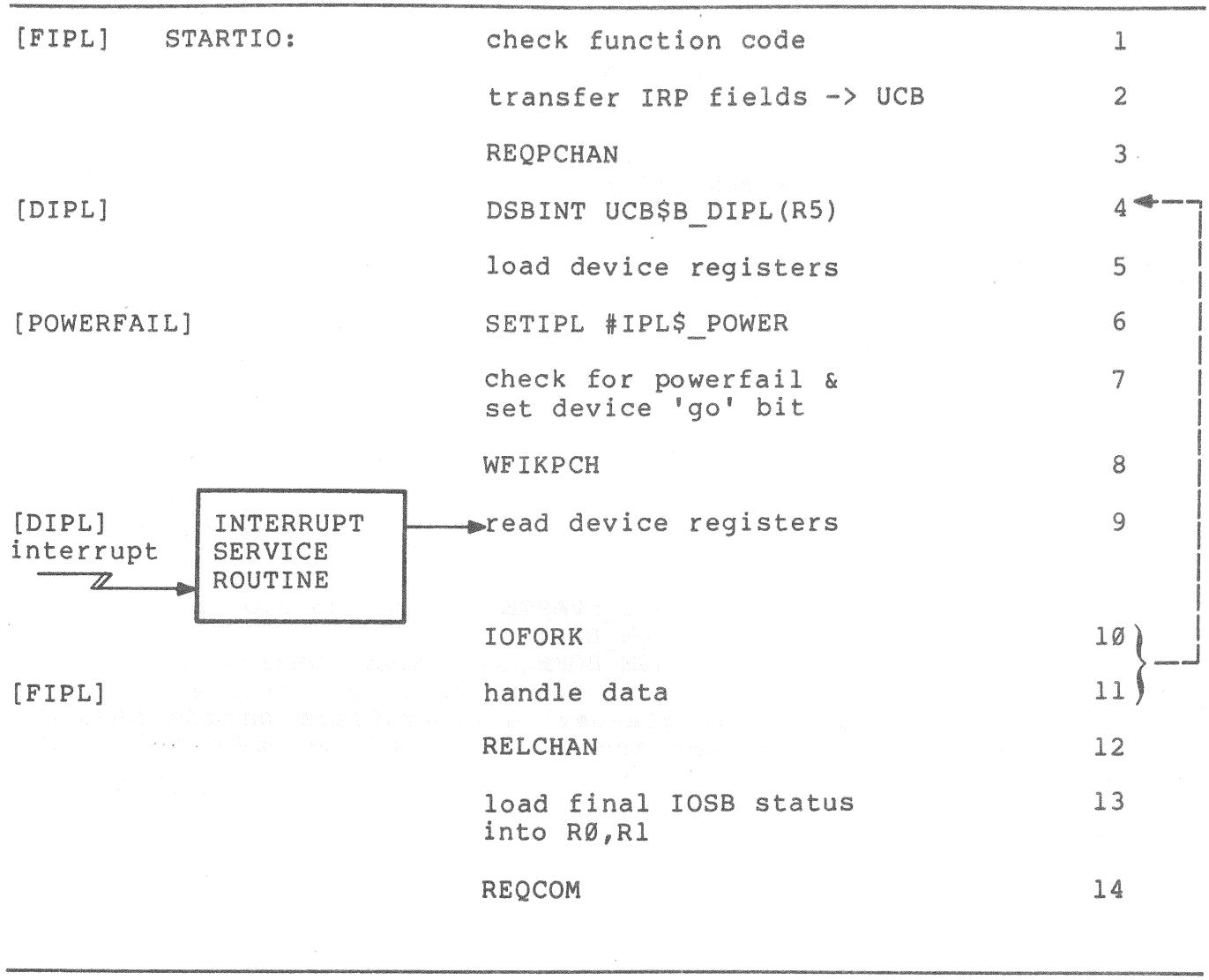
```
MOVL @IDB$L_ADP(R3),R5          ; R5 = MBA CSR
MOVAB MBA$L_ERB(R5),R2          ; R2 = BASE OF MBA REGISTERS
SUBL3 R2, R4, R2                ; DISTANCE TO CONTROLLER FROM
                                ; BASE OF MBA
ASHL #-7, R2, R2                ; R2 = MBA UNIT NUMBER
                                ; (DIVISION BY 128)
ASHL R2, #1, MBA$L_AS(R5)       ; CLEAR ATTENTION
                                ; SUMMARY BIT
```

If the driver wishes to use R0 and/or R1, these registers must be explicitly saved and restored.

MASSBUS drivers for single-unit controllers may specify an unsolicited interrupt routine address in the UNSOLIC field of the DDTAB macro. The unsolicited interrupt routine is called directly by MBA$INT (via a JSB instruction), and should exit via an RSB. Registers R2-R5 should **not** be restored, and an REI instruction should not be executed in an unsolicited interrupt service routine. When called (at device IPL), the unsolicited interrupt service routine can expect R4 to contain the CSR address for the controller, and R5 to contain the UCB address.

APPENDIX

Buffered I/O Case



The following notes are keyed to the buffered I/O case:

1. The function code may be found in IRP$W_FUNC(R3). Normally, instructions like the following are used:

```
CMPZV #IRP$V_FCODE, #IRP$S_FCODE,-
      IRP$W_FUNC(R3), #IO$_function_code
BEQL  10$       ;found function_code
```

   Read requests may be distinguished in another way. The FUNC bit is set by the system in the status word of the IRP.

```
BBS #IRP$V_FUNC, IRP$W_STS(R3), START_READ
```

2. Any fields your FDT routines wrote into the IRP, which your start I/O routine needs, should be transferred into the UCB (with appropriate MOVx instructions). Before your start I/O routine is called, the following fields are transferred for you:

```
IRP$L_SVAPTE -> UCB$L_SVAPTE
IRP$W_BCNT   -> UCB$W_BCNT
IRP$W_BOFF   -> UCB$W_BOFF
```

   Note that for single-device controllers on the MASSBUS, the IDB contains the address of the MBAs CSR. For multidevice controllers, the IDB contains the address of the controllers CSR. The UCB contains the unit number in UCB$B_SLAVE, and the index to the address of the first device register in UCB$B_SLAVE+1. These values are placed into the UCB by the unit initialization routine.

3. For single-unit controllers, this step is not necessary if you place the UCB address in the owner field of the IDB (see CRDRIVER unit initialization routine).

4. It is good practice to touch device registers at device IPL (although it is not required). It is required that a DSBINT macro be called before the WFIKPCH macro, since the DSBINT macro pushes an IPL on the stack, which the WFIKPCH macro removes. The IPL pushed on the stack should be FIPL, since that is the IPL at which the start I/O routine is called. General rule: exit from a routine at the same IPL with which you were called.

5. This is completely device-dependent. Sometimes bits are set in system registers (low-order byte/word). When all bits are set, an entire byte or word is written to the device register. MASSBUS drivers must clear any errors

in the MBA by setting -1 in the MBA status register (this is a write-ones-to-clear register). For example, if R4 contains the MBA CSR address, you can use the following instruction:

```
MCOML #Ø, MBA$L_SR(R4)
```

6. Before the device is started, a check is made to see if a powerfailure occurred since the time the first device register was loaded. Before making the check, IPL is raised to 31(IPL$_POWER) to block out all interrupts (including powerfail). A SETIPL macro is used, which does not place an IPL on the stack. If no DSBINT macro has previously been issued, a DSBINT macro must be used in place of a SETIPL macro.

7. Powerfailure is detected by the following instruction:

```
BBSC #UCB$V_POWER, UCB$W_STS(R5), POWERFAIL
```

Note that the driver must clear the powerfail bit (more on powerfail later). If power has not failed, then the 'go' bit of the device is set to initiate the hardware.

8. The WFIKPCH macro restores IPL (via an ENBINT macro) to a previous level, and returns control to the routine that started the I/O operation so that system processing may continue while the driver fork process waits for an interrupt or timeout. In the WFIKPCH call, specify the address of a timeout routine to be entered if a timeout occurs, and the number of seconds before a timeout should be declared. You must specify at least two seconds, and the timeout will occur ± one second from the time you specify. Note that a similar macro, WFIRLCH, may also be called. The only difference is that the WFIRLCH macro releases the controller channel (e.g., for a disk seek, in which no data transfer occurs), while the WFIKPCH macro retains ownership of the controller channel.

9. When a device interrupt occurs, the interrupt service routine is entered. It is the responsibility of the interrupt service routine to restore the fork process context, and transfer control to the stored PC, so that the next instruction after the WFIKPCH is executed. Since the interrupt service routine is entered at device IPL, the driver is resumed at device IPL. Typically, the driver only wants to stay at device IPL long enough to read device registers into fields in the UCB.

10. The IOFORK macro is used to lower IPL back to fork level (since the driver does not need to be at the higher device IPL). Note that only registers R3 and R4 will be the same after the IOFORK macro. (That is why you stored the contents of your device registers in fields in the UCB, and not in system registers.) If your device requires several interrupts to complete an I/O request, you may loop back to before the WFIKPCH macro (after decreasing some counter in the UCB). Make sure you loop back to somewhere that will issue a DSBINT macro before issuing the WFIKPCH macro.

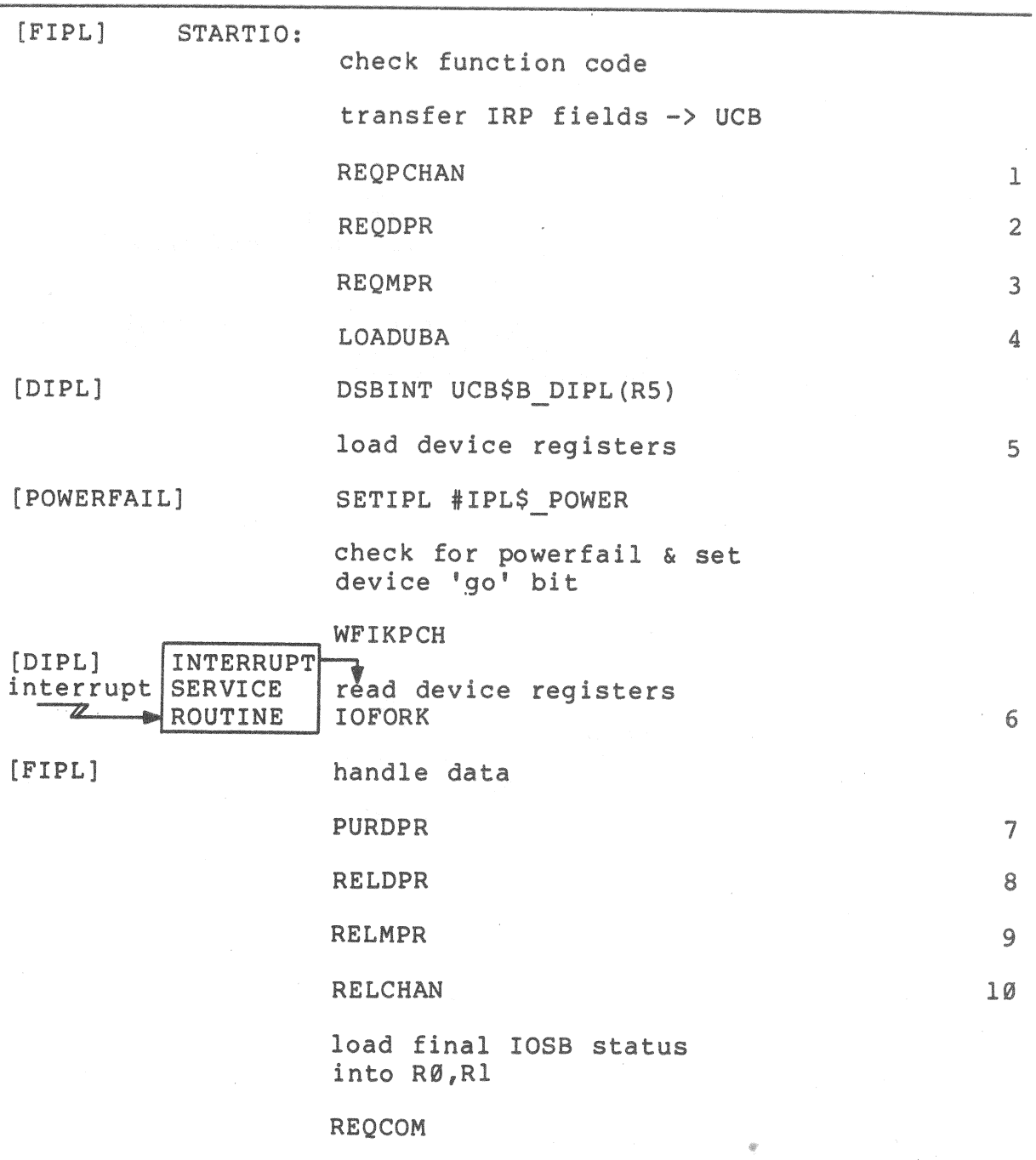
11. This is also completely device-dependent.

12. Do this only if you requested the channel in step 3.

13. If all is okay, load SS$_NORMAL into the low-order word of R0, and the number of bytes transferred into the high-order word (see CRDRIVER for an example). If you return any device-dependent data, place that in R1. If you do not return any device-dependent data, zero R1. If all is not okay with the I/O request, place the appropriate SS$_error_code in R0.

14. Your driver is finished. The rest of the I/O post-processing operation is handled by the system.

Direct I/O Case



The following notes are keyed to the Direct I/O case:

For those lines without numbers, see the notes for the buffered I/O case.

1. If you know you are not going to keep the channel very long (e.g., disk seek), you can call this macro with an argument PRI=HIGH, in which case you will be put at the front of the wait queue (if there is one) for the controller, rather than at the end, which is the default.

   For a data transfer on a multidevice controller on the MASSBUS, the driver must request both the primary controller (the device controller) and the secondary controller (the MASSBUS adapter). For single-device MASSBUS controllers, the primary controller is the MASSBUS adapter. The primary channel is always requested first, using REQPCHAN, after which the secondary channel is requested, if needed, using REQSCHAN. The secondary channel may also be requested with the PRI=HIGH argument, if desired. Nontransfer functions do not require the MBA (e.g., tape positioning functions require only the magnetic tape controller, and the unit); the MBA is free for other operations (e.g., data transfers on other units).

2. This is only necessary if

   - you have a UNIBUS device.
   - you want a buffered data path.
   - you have not permanently allocated a data path in your unit/controller initialization routines.

3. This is required for any DMA transfer. The system figures out how many mapping registers you need from the UCBL_SVAPTE, UCBW_BCNT, and UCB$W_BOFF fields. You get one more than you need (which the system marks as invalid) to avoid unwanted prefetches (discussed in more detail in the I/O Architecture module).

4. This is also required for any DMA transfer to load the previously allocated mapping registers. Macro LOADMBA is used by MASSBUS drivers.

   LOADMBA expects R4 to contain the address of the MBA CSR, and R5 to contain the address of the UCB. The routine preserves R3 but destroys RØ-R2. Along with loading the mapping registers, LOADMBA moves the negated value of the transfer byte count (UCB$W_BCNT) into MBA$L_BCR (the

internal MBA byte count register), and moves the byte offset in the first page (UCB$W_BOFF) into MBA$L_VAR (the internal MBA virtual address register).

5. Most DMA devices have two registers which require the following information:

   - how many bytes/words to transfer.
   - the starting (UNIBUS) address at which to begin the transfer.

   The first register is easy: just use the UCB$W_BCNT field (for bytes), or divide that value by 2 (for words). The calculation for the UNIBUS starting transfer address is somewhat trickier. One problem is that UNIBUS addresses are 18 bits big, while UNIBUS device registers are 16 bits big. Two bits, therefore, have to go into a second device register (typically, bits 7 and 8 of the CSR). These two bits are the high-order two bits, and are referred to as memory extension bits.

   The 18-bit UNIBUS address consists of two 9-bit fields (as shown in Figure 8-1):

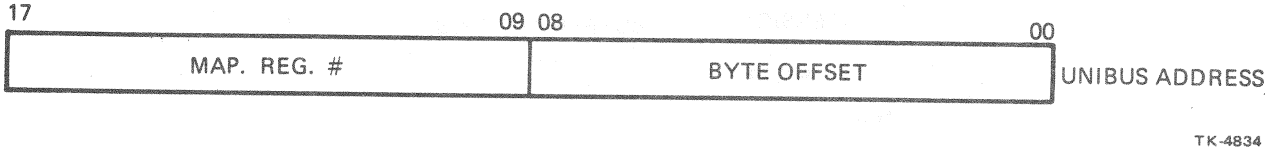


Figure 8-1 18-Bit UNIBUS Address

   Each mapping register maps one page of addresses (discussed in detail in the I/O Architecture module). Figure 8-2 shows how to construct the starting UNIBUS address.

6. For a DMA transfer, there is little need for a loop since the interrupt indicates that the transfer has completed. One interrupt is usually enough to satisfy an I/O request.

7. This is required to purge a buffered data path in the UNIBUS Adapter (discussed in the I/O Architecture module).

8. Only if step 2 was present.

9. Only if step 3 was present.

10. RELCHAN releases all controller channels (both primary and secondary). A MASSBUS driver that desires to release only the secondary controller channel can invoke the RELSCHAN macro.

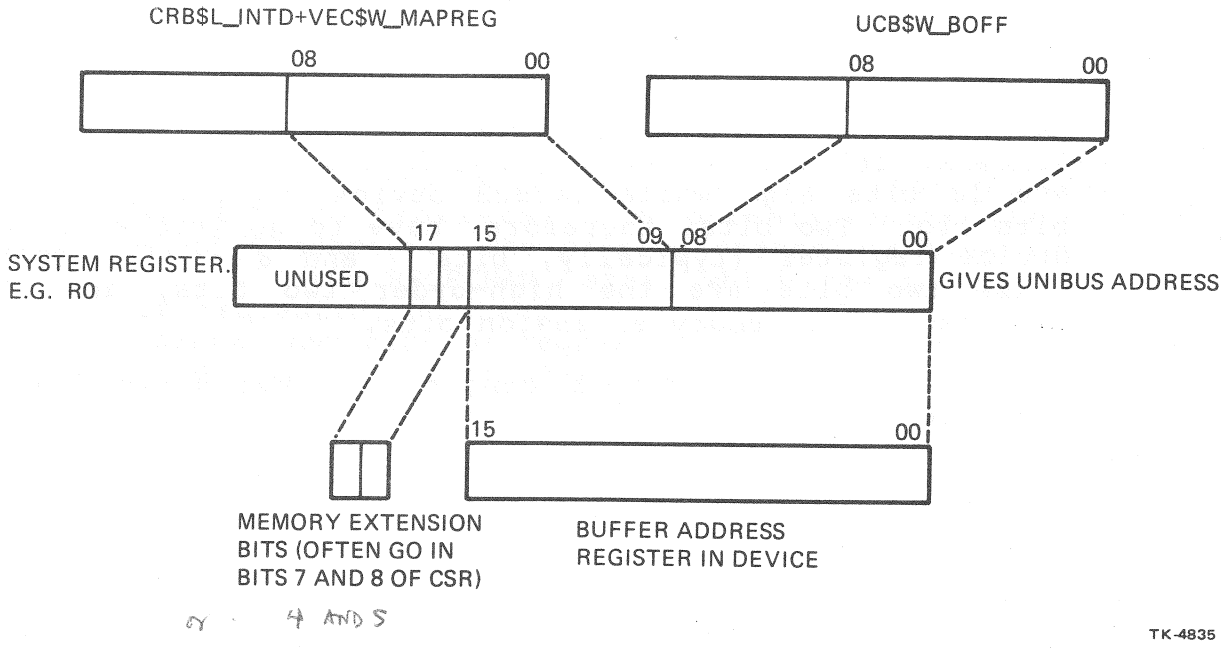


Figure 8-2 Construction of the Starting UNIBUS Address for a DMA Transfer

See Appendix on DR11s in the driver manual for the instructions that perform the above steps.

digital

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