Twenty Dollars

Cromemco®
Modifying A Cromix® Driver

CROMEMCO, Inc.
280 Bernardo Avenue
Mountain View, CA. 94043

Part No. 023-4052

February 1983

Cromemco®
Modifying A Cromix® Driver

CROMEMCO, Inc.
280 Bernardo Avenue
Mountain View, CA. 94043

Part No. 023-4052

February 1983

This manual was produced using a Cromemco System Three computer with a Cromemco HDD-22 Hard Disk Storage System running under the Cromemco Cromix® Operating System. The text was edited with the Cromemco Cromix Screen Editor. The edited text was proofread by the Cromemco SpellMaster™ Program and formatted by the Cromemco Word Processing System Formatter II. Camera-ready copy was printed on a Cromemco 3355B printer.

TABLE OF CONTENTS

Chapter 1

ADDING A NEW CHARACTER DEVICE DRIVER

To establish a Cromix system that includes a new driver that does not reside in the IOP, follow these instructions.

1. Write the new device driver. See the following pages for hints. It may be helpful to use the modules in the **/cro/driv/tty**, **/cro/driv/lpt**, and **/cro/driv/slpt** directories as models. As an example for this description, it has been assumed that a new driver is contained in two modules named **xyza.z80** and **xyzb.z80**, and that the first of these modules contains an entry point named **xyz**.

2. Assemble the modules of the new driver.

```
# asmb xyza
# asmb xyzb
```

3. Save the old copy of **cromix.sys**. Use the Crogen utility to create a copy of **cromix.sys** that includes the new driver.

```
# move /cromix.sys /cromixold.sys
# crogen -u /cromix
```

First the Crogen utility asks, one by one, which of the standard Cromix drivers should be included. After these questions have been answered, it will ask questions about the new driver. Major device numbers for user-supplied drivers range from 20 up. These questions and answers are illustrated for the driver XYZ with device number 21.

```
User supplied character drivers (Y=Yes, N=No) <N> y

Enter driver entry point (blank=none, E=End)

          20 -
          21 - xyz
          22 - e

Enter names of rel files which contain drivers (E=End)

          ?  xyza
          ?  xyzb
          ?  e
```

4. Run the Makdev utility to define the devices that will use the driver. The Makdev utility does this by assigning the device names to the corresponding device numbers. For example, three devices could be defined by the following commands, assuming that the major device number is 21.

```
# makdev /dev/xyz1 c 21 0
# makdev /dev/xyz2 c 21 1
# makdev /dev/xyz3 c 21 2
```

6. Boot the new operating system.

```
boot
```

WHAT IS A DEVICE DRIVER?

A character device driver is a list of the addresses of seven subroutines, called the **primary driver subroutines.** A driver can also have other subroutines. The list of primary subroutines for the xyz device driver takes the following form:

```
        entry   xyz

xyz:    dw      init                ; initialize device
        dw      open                ; open device
        dw      close               ; close device
        dw      read                ; read device
        dw      write               ; write to device
        dw      getmode             ; get device mode
        dw      setmode             ; set device mode
```

The primary driver subroutines must satisfy only a few requirements. Many drivers in effect omit primary subroutines by providing dummy subroutines that do nothing but return.

WHEN ARE THE PRIMARY DRIVER SUBROUTINES CALLED?

The first subroutine, the Initialize subroutine, is called once, when the Cromix Operating System is booted.

The Open subroutine is called whenever the device is opened by the **.open** system call.

The Close subroutine is called whenever the device is closed by the **.close** system call.

The Read subroutine is called whenever the device is read by the **.rdseq, .rdline,** or **.rdbyte** system calls.

The Write subroutine is called whenever anything is written to the device by the **.wrseq, .wrline,** or **.wrbyte** system calls.

The Getmode subroutine is called whenever the mode of the device is examined by the **.getmode** system call.

The Setmode subroutine is called whenever the mode of the device is set by the **.setmode** system call.

HOW DO THE PRIMARY DRIVER SUBROUTINES WORK?

Following is a brief description of the primary subroutines for drivers that use interrupts. Driver interrupt subroutines are also described. Reference is made to various utility subroutines that are subsequently described in detail.

Primary driver subroutines must **not** make Cromix system calls.

Initialize Subroutine

When the Cromix Operating System is booted, the Initialize subroutine for each device driver is called. Interrupts must remain disabled during this time. In some drivers, this subroutine initializes the interface board. Drivers for devices that use the Tuart interface can use the Tuinit utility for this purpose. Drivers that use the Quadart can use the Qdinit utility.

Open Subroutine

This subroutine is called when a **.open** system call is made. Many drivers use this subroutine to initialize a data structure for the device and set up the device's interrupts the first time the device is opened. The Cromix Operating System maintains a set of data structures for devices that use the Tuart interface. The Tussetup utility finds a data structure for a serial Tuart device, initializes it, and sets up its interrupts. The Cromix Operating System also maintains a set of data structures in the IOP for devices that use the IOP and Quadart interface. The Qdsetup utility sets up these devices.

Close Subroutine

This subroutine is called when a **.close** system call is made. It is used by many drivers to relinquish the data structure and mask the interrupts when the device is finally closed. The Tusshut and Qdshut utilities perform these functions for serial Tuart devices and IOP-Quadart devices, respectively.

Read Subroutine

This subroutine is called when a **.rdseq, .rdline,** or **.rdbyte** system call is made. If there are not enough characters in the device input queue, the Read subroutine should call the Waitid utility to obtain a wait identification number and then call the Wait utility. The wait identification number specifies the reason for the wait. If the device has a character available, the Read subroutine should retrieve it from the input queue by calling the Getq utility and then deliver it to the system by calling the Ioputc utility. If Ioputc returns with the carry flag reset, this entire procedure should be repeated. Otherwise, the system call has been satisfied and the Read subroutine should return with the carry flag reset.

Input Interrupt Subroutine

Assuming that the Open subroutine (above) has set up interrupts using either the Tussetup, Qdsetup, or Addint utility, the interrupt handler will branch to this subroutine when the device hardware causes an input interrupt. This subroutine should read a character or characters from the device and place them in the input queue by calling the Putq utility. If the driver's read subroutine has put the process to sleep by calling Wait, then the interrupt subroutine should wake the process up by calling the Waitid and Wakeup utilities after the input queue has been appropriately filled. When the queue is appropriately filled will depend how the device is to be used. Terminal drivers, for example, usually do not wake the read process up until the queue contains a full line.

Write Subroutine

This subroutine is called when a **.wrseq**, **.wrline**, or **.wrbyte** system call is made. If the number of characters already in the output queue exceeds a specified number, the Write subroutine should call the Waitid utility to obtain a wait identification number and then call the Wait utility. The wait identification number specifies the reason for the wait. Otherwise, the Write subroutine should call the Iogetc utility to get a character from the system. If Iogetc returns with the carry flag reset, the Write subroutine should deliver the character to the output queue by calling the Putq utility and then repeat this entire procedure. If the carry flag is set, the system has no more characters to deliver and the Write subroutine should return with the carry flag reset.

Output Interrupt Subroutine

Assuming that the Open subroutine (above) has set up interrupts using either the Tussetup, Qdsetup, or Addint utility, the interrupt handler will branch to this subroutine when the device hardware causes an input interrupt. This subroutine should get a character from the output queue by calling the Getq utility and then write it to the device. If the Write subroutine has put the process to sleep by calling Wait, then the interrupt subroutine should wake the process up by calling the Waitid and Wakeup utilities after the number of characters in the output queue has fallen to a specified number.

Getmode Subroutine

This subroutine is called when a **.getmode** system call is made. Its function is left up to the driver.

Setmode Subroutine

This subroutine is called when a **.setmode** system call is made. Its function is left up to the driver.

HOW DO THE PRIMARY DEVICE SUBROUTINES USE THE Z80 REGISTERS?

When the Cromix Operating System calls any of the primary device subroutines, the **de** register contains the device number. (The major device number is in the **d** register and the minor number is in the **e** register.) The major device number is assigned to the driver by the Crogen utility. The minor number is used at the discretion of the driver. Many drivers use it to correspond to device port addresses.

A primary device subroutine should return to the Cromix Operating System with the carry flag set to indicate an error condition. Otherwise, the carry flag should be reset. If the carry flag is set the **a** register should contain a number indicating the nature of the error. This number should be chosen from one of the standard error numbers defined in the **jsysequ.z80** file. For example, if the open routine is called with an inappropriate device number, the routine should return with the carry flag set and with the **a** register containing the constant **?devopen.**

The subroutines for getting and setting the mode are called with the **c** register containing the contents of the **c** register when the **.getmode** or **.setmode** system call was made. The **hl** register contains the contents of the **de** register when the system call was made. When one of these subroutines returns, the contents of the **hl** register will appear in the **de** register when the system call returns. This provides a means of passing parameters to these subroutines and returning values from them.

With these exceptions, no registers have significance upon return from any of the primary driver subroutines.

The Initialize subroutine is called with interrupts disabled. It must **not** enable interrupts. The other six subroutines are called with interrupts enabled. These subroutines may disable interrupts during critical sections, but otherwise interrupts should be left enabled.

Some examples of drivers follow. These drivers make use of various Cromix driver utilities, such as Ioputc and Iogetc. When one of these utilities is used, its function is hinted at in the comments. Detailed descriptions are provided following the listings.

Example 1

```
                entry   ttyni
                ext     tuinit
                ext     ioputc, iogetc

data    equ     21h                 ; data port number
stat    equ     dataport - 1        ; status port number

ttyni:  ; An elementary non-interrupting terminal driver
        ; which uses the serial Tuart port at address 21H
        dw      tuinit              ; Initialize
        dw      dummy               ; Open
        dw      dummy               ; Close
        dw      niread              ; Read
        dw      niwrite             ; Write
        dw      dummy               ; Get mode
        dw      dummy               ; Wet mode

dummy:  xor     a,a                 ; Clear the carry flag
        ret

niread:
rd0:    in      a,(stat)            ; Wait until the serial input
        bit     6,a                 ; port is ready
        jr      z,rd0
        in      a,(data)            ; and then input the byte
        call    ioputc              ; Deliver the byte to process
                                    ; which made the .RDSEQ, .RD-
                                    ; LINE, or .RDBYTE system call
        jr      nc,rd0              ; Are more bytes wanted?
        ; No.  The system call has been satisfied
        and     a,a                 ; Clear carry flag
        ret

niwrite:
wr0:    call    iogetc              ; Get a byte from the process
                                    ; which made the .RDSEQ, .RD-
                                    ; LINE, or .RDBYTE system call
        jr      c,wrexit            ; Was a byte available?
wr2:    in      b,(stat)            ; Yes.  Wait until the serial
        bit     7,b                 ;       output port is ready
        jr      z,wr2
        out     (data),a            ; Output the byte
        jr      wr0                 ; Loop until no more bytes
                                    ; are available from the system
wrexit: ; Done.  The system call has been satisfied
        and     a,a                 ; Clear carry flag
        ret
```

Example 2

```
        entry   eslpt
        ext     ioputc, iogetc, putq, getq
        ext     waitid, wait, wakeup
        ext     tuinit, tussetup, tusfind, tusshut, tuspeed

*include jsysequ.z80
*include modeequ.z80
*include waitdef.z80

eslpt:  ; An elementary interrupting serial printer driver
        ; which uses the serial Tuart port whose address
        ; is determined by the minor device number
        dw      tuinit          ; Initialize
        dw      eopen           ; Open
        dw      eclose          ; Close
        dw      dummy           ; Read
        dw      ewrite          ; Write
        dw      dummy           ; Get mode
        dw      dummy           ; Set mode

; ----------------------------------------------------------
; Definition of the data structure for this driver

        struct  0
devno:    ds    2       ; Device number
dataprt:  ds    1       ; Data port number
openct:   ds    1       ; Open count
stat:     ds    1       ; Status
outq:     ; Header for the output queue
          ds    1       ; Output queue byte count
          ds    2       ; Space for two queue pointers
          ds    2
        mend

; STAT bits
WRWAIT  equ     0       ; Waiting for output queue to drain
INT_DUE equ     1       ; An output interrupt is expected

; ----------------------------------------------------------
; Constants

OUTHIGH equ     60      ; High-water mark for output queue
OUTLOW  equ     30      ; Low-water mark for output queue

; ----------------------------------------------------------
; Dummy subroutine

dummy:  xor     a,a             ; Clear the carry flag
        ret

; ----------------------------------------------------------
; Open subroutine

eopen:  ; Called by Cromix with DE = device number
        di
        ; Set up the serial Tuart for this device.
        ; This includes setting up interrupts by informing
        ; Cromix of the addresses of the input and the output
        ; interrupt routines for this driver.
        ld      bc,dummy        ; Input interrupt routine
        ld      hl,outint       ; Output interrupt routine
        call    tussetup
        jr      c,opx2          ; Error?
        ; No.  TUSSETUP returns with both DE and IY containing
        ; the address of the data structure for this device
        ; (determined by the minor device number).  It also
        ; sets up the interrupt handler to load DE with the
        ; address of the data structure when the interrupt
        ; routines are called.
        jr      nz,op10         ; Is this the first open?
        ld      (iy+openct),1   ; Yes.  Open count = 1
        ld      a,S_1200        ; Code for 1200 baud
        call    tuspeed         ; Set the Tuart baudrate
opx1:   and     a,a             ; Clear carry flag
opx2:   ei
        ret

op10:   ; The device has been opened before
        inc     (iy+openct)     ; Update the open count
        jr      nz,opx1         ; Too many opens?
        dec     (iy+openct)     ; Yes
        ld      a,?devopen      ; Error
        scf
        jr      opx2            ; Exit
```

```
; -----------------------------------------------------------
; Close subroutine

eclose: ; Called by Cromix with DE = device number
        ld      c,e             ; Save minor device number
        ; Find the serial Tuart data structure for this device
        call    tusfind         ; DE & IY = addr of structure
        di
        dec     (iy+openct)     ; Update the open count
        jr      nz,clx          ; Last open?
        call    oqflush         ; Yes.  Flush the output queue
        ld      e,c             ; E = minor device number
        call    tusshut         ; Shut down the device
clx:    and     a,a             ; Reset carry flag
        ei
        ret

oqflush:; Discard anything remaining in the output queue
        ; Call with:    DE = address of device structure
        push    hl
         ld     hl,outq         ; Displacement of output queue
         add    hl,de           ; HL = addr of output queue
f12:     call   getq
 jr     nc,f12
        pop     hl
        ret
```

```
; -----------------------------------------------------------
; Write subroutine

ewrite: ; Called by Cromix with DE = device number
        ; Find the serial Tuart data structure for this device
        call    tusfind         ; DE & IY = addr of structure
wr0:    ; Get a byte from the process which made the .RDSEQ,
        ; .RDLINE, or .RDBYTE system call
        call    iogetc
        jr      c,wrexit        ; Was a byte available?
        ld      h,a             ; Yes.  Save it
wr2:    ; Get count of number of bytes already in output queue
        ld      a,(iy+outq)
        cp      a,OUTHIGH       ; Too many?
        jr      c,wr8
        ; Yes.  Wait until outp. queue has drained sufficiently
        set     WRWAIT,(iy+stat)
        ld      a,HmWRWAIT      ; Code for the reason for wait
        ld      l,(iy+devno)    ; Minor device number
        ld      h,(iy+devno+1)  ; Major device number
        call    waitid          ; Calculate the wait ID number
        call    wait            ; Wait until awaken
        jr      wr2             ; When awakened, check again
wr8:    ; Put the byte into the output queue
        ld      hl,outq         ; Displacement of output queue
        add     hl,de           ; HL = address of output queue
        call    putq
        ; If no output interrupt is due, call the output
        ; interrupt routine in order to output a byte.
        bit     INT_DUE,(iy+stat); Is an interrupt expected?
        call    z,outint2
        ; Loop until no more bytes are available from system
        jr      wr0
wrexit: ; Done.  The system call has been satisfied
        and     a,a             ; Clear carry flag
        ret
```

```
; -------------------------------------------------------------
; Output interrupt subroutine

outint: ; Output interrupt subroutine
        ; Called with DE = address of the device structure
        push    de
        pop     iy                  ; DE and IY = addr of dev struc
outint2:ld      hl,outq             ; Displacement of output queue
        add     hl,de               ; HL = address of output queue
        call    getq                ; Get a byte from it
        jr      c,oul0              ; Got one?
        ld      c,(iy+dataprt)      ; Yes.  Get addr of data port
        out     (c),a               ;       and output the byte
        ; Since an output has just been made, we can expect
        ; another interrupt
        set     INT_DUE,(iy+stat)
        ; Awaken the waiting Cromix process, if appropriate
        call    wrwake?
        ret                         ; Done

oul0:   ; Nothing was output from the queue, so no further
        ; interrupts will be forthcoming
        res     INT_DUE,(iy+stat)
        ret

wrwake?:; Awaken Cromix process if appropriate
        bit     WRWAIT,(iy+stat); Is it asleep (i.e., waiting)?
        ret     z               ; If not, done
        ; Yes.  Get count of number of bytes in output queue
        ld      a,(iy+outq)
        cp      a,OUTLOW            ; Is it at the low-water mark?
        jr      z,ww2
        and     a,a                 ; No.  Is it empty?
        ret     nz
ww2:    ; Ready for more output, so awaken the Cromix process
        res     WRWAIT,(iy+stat)
        ld      a,HmWRWAIT          ; Code for the reason for wait
        ld      l,(iy+devno)        ; Minor device number
        ld      h,(iy+devno+1)      ; Major device number
        call    waitid              ; Calculate the wait ID number
        call    wakeup              ; Wake process up
        ret
```

UTILITIES FOR ADDING AND REMOVING INTERRUPT VECTORS

Most character drivers use interrupts. The following subroutines provide drivers a means of adding interrupt vectors to, or removing them from, the system.

The Tussetup and Qdsetup utilities can also be used to set interrupts up. They call Addint in order to do this.

Addint

Call Addint with the Z80 Mode-2 interrupt vector in the **a** register, the address of the interrupt subroutine in the **bc** register, and data which will be loaded into the **de** register whenever the interrupt occurs in the **de** register.

Addint returns with the carry flag set if that interrupt vector has already been used. Otherwise, it returns with the carry flag reset.

Addint preserves all registers except the **af** register.

Subint

Call Subint with the Z80 Mode-2 interrupt vector in the a register.

Subint preserves all registers except the **af** register.

CHARACTER TRANSFER UTILITIES

The Cromix Operating System provides system calls for reading and writing one or more characters from or to a device. A driver can deliver the characters requested by a read system call by repeatedly calling the subroutine Ioputc until it returns with the carry flag set. Similarly, a driver can get the characters being sent by a write system call by repeatedly calling the subroutine Iogetc until it returns with the carry flag set.

Ioputc

Call Ioputc with the character to be sent to the system in the a register.

Ioputc returns with the carry flag reset if the system will accept at least one more character. Otherwise, it returns with the carry flag set.

Ioputc preserves all registers except the **af** register.

Iogetc

If there is a character available from the system, Iogetc returns with it in the **a** register and with the carry flag reset. Otherwise, it returns with the carry flag set.

Iogetc preserves all registers except the **af** register.

CHARACTER BUFFERING UTILITIES

A character driver that uses interrupts must buffer the characters it processes unless the device is fast enough to handle the characters at the rate the system can send them. A driver can use the Putq and Getq utilities to buffer characters in a FIFO system character queue. The space to store the characters is provided by the operating system. The driver needs to provide five bytes of storage called the queue header which Putq and Getq use to keep track of the queue. The queue header consists of a byte count and two pointers. The driver must initialize the byte count to zero. After this, Putq and Getq handle the byte count and the queue pointers.

An example of a header for a queue named Outq follows.

```
outq:   ds      1       ; byte count
        ds      2       ; pointer
        ds      2       ; pointer
```

Putq

Call Putq with the address of the queue header in the **hl** register and the character to be put into the queue in the **a** register.

Putq returns with the carry flag set if there was no space to store the character. Otherwise, it returns with the carry flag reset.

Putq preserves all registers (including the **af** register).

Getq

Call Getq with the address of the queue header in the **hl** register.

Getq returns with the carry flag set if the queue is empty. Otherwise, it returns with the carry flag reset and the character in the **a** register.

Getq preserves all registers except the **af** register.

UTILITIES FOR SLEEPING AND WAKING UP

When a process calls a driver to ask for a resource which the driver does not currently have, it should suspend the process by calling the Wait utility. When the resource becomes available the Wakeup utility should be called so that the process can continue execution. Wakeup is usually called by an interrupt subroutine. For example, a read system call from a driver which has no characters in its input queue should result in a call to Wait. Wakeup should be called by the input interrupt subroutine whenever the specified number of characters have been placed into the input queue.

Wait

Call Wait with an identification number in the **hl** register. This number serves to identify the reason for the wait.

Wait returns when the process has been woken up by a call to Wakeup with the same identification number.

Wakeup

Call Wakeup with the same identification number in the **hl** register which was used when the call to Wait was made.

Wait and Wakeup preserve all registers except the **af** register.

A standardized wait identification number can be obtained by a call to Waitid.

Waitid

Call Waitid with the device number in the **hl** register and a constant in the **a** register which specifies the reason for the wait. The following constants are defined in **waitdef.z80**.

HmRDWAIT	wait until input is ready
HmWRWAIT	wait until the output queue is sufficiently empty
HmSCWAIT	wait for a start character (X-ON or CNTRL-Q)

Waitid returns with the wait identification number in the **hl** register.

UTILITIES FOR DRIVERS THAT USE THE TUART INTERFACE

Tuinit

This utility initializes all the Tuart interface boards. It should be specified as the Initialize subroutine for all drivers that use the Tuart interface board.

Call Tuinit with interrupts disabled.

Tuinit alters the **af** register.

Tussetup

This utility sets up a serial Tuart device driver. It obtains a data area for the device data structure, initializes it, and adds the input and output interrupt vectors to the Z80 interrupt page.

Call Tussetup with interrupts disabled, the device number in the **de** register, the address of the input interrupt routine in the **bc** register, and the address of the output interrupt routine in the **hl** register.

If no error, this utility finds the address of the device data structure (determined by the minor device number), initializes it, sets up the interrupt handler so that it will load the **de** register with the address of the data structure when the interrupt routines are called, and returns with the carry flag reset and with the address of the device data structure in both the **de** and the **iy** registers. The zero flag is set if this is the first time the device was opened.

If the device number is illegal or if there is no data area available for the device data structure, Tussetup returns with the carry flag set and an error number in the **a** register.

Tussetup alters the **af**, **bc**, **de**, **hl**, and **iy** registers.

The Tussetup utility calls the following utilities: Tuscheck, Tusfind, Tustruct, Tuioints, Tudport, and Tuunmask.

Tusshut

This utility shuts down a serial Tuart device driver. It removes the interrupts from the Z80 interrupt page and relinquishes the data area.

Call Tusshut with the minor device number in the **E** register.

Tusshut alters the **af**, **bc**, **de**, **hl**, and **iy** registers.

The Tusshut utility calls the following utilities: Tusfind, Tuioxints, and Tumask.

Tupshut

This utility shuts down a parallel Tuart device driver. It removes the interrupts from the Z80 interrupt page and relinquishes the data area.

Call Tupshut with the minor device number in the **e** register.

Tupshut alters the **af**, **bc**, **de**, **hl**, and **iy** registers.

The Tupshut utility calls the following utilities: Tupfind, Tupxint, and Tumask.

Tuscheck

This utility checks whether the minor device number for a serial Tuart device is legal.

Call Tuscheck with the minor device number in the **a** register.

Tuscheck returns with the carry flag set and the constant **?nodevice** in the **a** register if the minor device number is illegal. Otherwise, the carry flag is reset.

Tuscheck alters the **af** and **hl** registers.

Tupcheck

This utility checks whether the minor device number for a parallel Tuart device is legal.

Call Tupcheck with the minor device number in the **a** register.

Tupcheck returns with the carry flag set and the constant **?nodevice** in the **a** register if the minor device number is illegal. Otherwise, the carry flag is reset.

Tupcheck alters the **af** and **hl** registers.

Tustruct

This utility finds space in the Cromix reserved data area for the data structure for a Tuart device.

Tustruct returns with the carry flag set if no space is available. Otherwise, the carry flag is reset and both the **de** and the **iy** registers contain the address of the data structure.

Tustruct alters the **af, id,** and **iy** registers.

Tusfind

This utility finds the address of the data structure assigned a serial Tuart device.

Call Tusfind with the device number in the **de** register.

Tusfind returns with the address of the device data structure in both the **de** and the **iy** registers.

Tusfind alters the **af, de, hl,** and the **iy** registers.

Tupfind

This utility finds the address of the data structure assigned a parallel Tuart device.

Call Tupfind with the device number in the **de** register.

Tupfind returns with the address of the device data structure in both the **de** and the **iy** registers.

Tupfind alters the **af, de, hl,** and the **iy** registers.

Tusdport

This utility gets the data port address of a serial Tuart device.

Call Tusdport with the minor device number in the **a** register.

Tusdport returns with the data port address in the **a** register.

Tusdport alters the **af** register.

Tuioints

This utility sets up the input and output interrupts for a serial Tuart device.

Call Tuioints with interrupts disabled, the input interrupt routine address in the **bc** register, the output interrupt routine address in the **hl** register, the address of the device's data structure in the **de** register, and the minor device number in the **a** register.

Tuioints returns with the address of the device data structure in both the **de** and the **iy** registers.

Tuioints alters the **af** register.

The Tuioints utility calls the Addint utility to do its work.

Tupint

This utility sets up the output interrupt for a parallel Tuart device.

Call Tupint with interrupts disabled, the output interrupt routine address in the **bc** register, the address of the device's data structure in the **de** register, and the minor device number in the **a** register.

Tupint returns with the address of the device data structure in both the **de** and the **iy** registers.

Tupint alters the **af** register.

The Tupint utility calls the Addint utility to do its work.

Tuioxint

This utility removes the input and output interrupts of a serial Tuart device from the Z80 interrupt page.

Call Tuioxint with interrupts disabled and the minor device number in the **a** register.

Tuioxint alters the **af** register.

The Tuioxint utility calls the Subint utility to do its work.

Tupxint

This utility removes the output interrupt of a parallel Tuart device from the Z80 interrupt page.

Call Tupxint with interrupts disabled and the minor device number in the **a** register.

Tupxint alters the **af** register.

The Tupxint utility calls the Subint utility to do its work.

Tuunmask

This utility unmasks Tuart interrupt(s).

Call Tuunmask with interrupts disabled, the Tuart data port address in the **c** register, and the **b** register containing ones in the bits corresponding to Tuart interrupts which are to be **unmasked.** (If a bit in the **b** register is zero, the corresponding Tuart interrupt will not be changed.)

Tuunmask alters the **af** register.

Tumask

This utility masks Tuart interrupt(s).

Call Tumask with interrupts disabled, the Tuart data port address in the **c** register, and the **b** register containing ones in the bits corresponding to Tuart interrupts which are to be **masked.** (If a bit in the **b** register is zero, the corresponding Tuart interrupt will not be changed.)

Tumask alters the **af** register.

Tuunmtbe

This utility unmasks Tuart serial output interrupt (TBE).

Call Tuunmtbe with interrupts disabled and the Tuart data port address in the **c** register.

Tuunmtbe alters the **af** register.

Tumsktbe

This utility masks Tuart serial output interrupt (TBE).

Call Tumsktbe with interrupts disabled and the Tuart data port address in the **c** register.

Tumsktbe alters the **af** register.

Tuspeed

This utility sets the baudrate of a serial Tuart device.

Call Tuspeed with the address of the device data structure in the **iy** register, the data port number in the **c** register, and the desired speed code in the **b** register. (The speed codes are defined in the file modeequ.z80.)

Tuspeed returns with the carry flag set if the speed code is illegal. Otherwise, returns with the carry flag reset.

Tuspeed alters the **af** register.

Tuconnect

This utility unmasks the serial input interrupt (RDA) if a terminal is connected to a serial Tuart device.

Call Tuconnect with the data port number in the **c** register.

Tuconnect returns with the carry flag reset and the input interrupt (RDA) unmasked if a terminal is connected. Otherwise, returns with the carry flag set.

Tuconnect alters the **af** register.

Chapter 2

ADDING A NEW IOP CHARACTER DEVICE DRIVER

The IOP contains a sizable memory on the C-bus which is independent of the main memory on the S-100 bus. This memory can be used to store Cromix drivers for devices that use C-bus interface boards such as the Quadart or CSP.

Putting a driver on the C-bus releases Cromix buffer space in the main S-100 bus memory. Another benefit stems from the fact that the IOP contains a small auxiliary operating system which works in tandem with the main Cromix Operating System on the S-100 bus. This IOP operating system only disables interrupts for a very short period of time. Therefore, IOP driver interrupts, such as keyboard interrupts in a terminal driver, are never lost.

Cromix drivers for devices on the IOP C-bus are usually composed of (1) a small interface driver in the Cromix Operating System on the host machine, and (2) the device driver proper that resides in the IOP on the C-bus.

Most C-bus Cromix drivers can be written so that they reside almost entirely in the IOP memory. They require only a small interface driver on the S-100 bus that is used to interface to the main driver in the IOP. Most C-bus drivers will work either with an S-100 interface driver called **iodev** or one called **iodevb.** For example, device number 2 and device number 9 both use **iodev.** Yet device number 2 is the C-bus terminal driver, **qtty** or **mtty,** and device number 9 is the C-bus serial printer driver, **qslpt.** The tape driver **tp** uses interface driver **iodevb.** The characteristics of **iodev** and **iodevb** are described later.

Configuring the Cromix Operating System to include a new IOP driver requires the use of the Crogen utility to add the Cromix-to-IOP interface driver to **cromix.sys.** The Crogen utility is located in the **/gen** directory. It also requires the Iopgen utility to add the C-bus driver to an IOP driver file in the **/dev/iop** directory. The Iopgen utility is located in the **/cro/driv/iop/gen** directory.

One example of an IOP driver file is **cromix.iop,** which contains the drivers for **qtty, mtty, qslpt,** and **iomem** devices. Another example is **tape.iop,** which contains the drivers for **tp** tape devices and for **iomem.** IOP driver files are loaded into the IOP by the command file **/etc/iostartup.cmd** when the Cromix Operating System is booted.

ADDING THE INTERFACE DRIVER TO CROMIX

Drivers that reside in the IOP on the C-bus require a Cromix-to-IOP interface driver that resides in **cromix.sys** in S-100 bus memory. Most of these drivers can use either **iodev** or **iodevb.** To incorporate such a driver, skip to step 3 below and use the name **iodev** or **iodevb** in place of **xyzintf.** If it is necessary to write a new interface driver, perform all four of the following steps.

1. Write the Cromix-to-IOP interface driver. (Use the modules in the directory **/iop/host** as models.) For illustration, assume that the interface for a new IOP driver is contained in two modules named **xyzintfa.z80** and **xyzintfb.z80,** and that the first of these modules contains an entry point named **xyzintf.**

2. Assemble the modules of the new interface driver.

```
# asmb xyzintfa
# asmb xyzintfb
```

3. If there is room on the disk, save the old copy of **cromix.sys.** Use the Crogen utility to create a copy of **cromix.sys** that includes the new interface driver.

```
# move /cromix.sys /cromixold.sys
# crogen -u /cromix
```

First the Crogen utility asks, one by one, which of the standard Cromix drivers should be included. After these questions have been answered, it will ask questions about the new driver. Major device numbers for user-supplied drivers range from 20 up. These questions and answers are illustrated for the interface driver XYZINF with device number 22.

```
User supplied character drivers (Y=Yes, N=No) <N> y

Enter driver entry point (blank=none, E=End)


     20 -
     21 -
     22 - xyzint
     23 - e


Enter names of rel files which contain drivers (E=End)


     ?  xyzinta
     ?  xyzintb
     ?  e
```

ADDING THE C-BUS DRIVER TO THE IOP DRIVER FILE

1. Write the driver that will reside in the IOP. (Use the modules in the subdirectories of **/cro/driv/iop/qd** and **/cro/driv/iop/csp** as models.) For illustration, assume that the new driver is contained in two modules named **xyza.z80** and **xyzb.z80**, and that the first of these modules contains an entry point named **xyz.** Also assume that the driver is to be added to an IOP driver file called **iopdriv.iop** in the directory **/dev/iop.**

2. Assemble the modules of the driver.

```
# asmb xyza
# asmb xyzb
```

3. Combine the driver modules into a library. Name the library after the driver entry point, **xyz.**

```
# lib xyz=xyza,xyzb/e
```

4. If there is room on the disk, save the old copy of **iopdriv.iop.** Use the Iopgen utility that is located in the directory **/cro/driv/iop/gen** in order to create a new copy of the **iopdriv.iop** that includes the **xyz** driver.

```
# move /dev/iop/iopdriv.iop /dev/iop/iopdrivold.iop
# /cro/driv/iop/gen/iopgen
```

The Iopgen utility asks whether a new driver is to be added. Answer **yes** to this question. It then asks for the name of the IOP driver file. The proper answer in the current example is **iopdriv.iop.** The Iopgen utility will ask whether the interface card to be used with the IOP is a Quadart or CSP, and it will prompt for the major device number and the name of the new device driver. Its major device number can be any unused number greater than 19. The name of the new driver is name of the driver entry point, **xyz.** If there are no more new drivers to be added, answer **no** the next time the Iopgen utility asks whether another driver is to be added.

Using the New Driver

1. Run **makdev** to define the devices that will use the driver by assigning the device names to the corresponding device numbers. If the major device number is 22, three devices could be defined as follows.

   ```
   # makdev /dev/xyz1 c 22 0
   # makdev /dev/xyz2 c 22 1
   # makdev /dev/xyz3 c 22 2
   ```

2. Make sure that **/etc/iostartup.cmd** includes a command to load the IOP driver file into the IOP. The following command loads the IOP driver file **iopdriv.iop** into IOP2.

   ```
   # /dev/iop/ioprun /dev/iop/iopdriv iop2
   ```

3. Boot the new operating system.

   ```
   boot
   ```

The Cromix-to-IOP Interface Driver

An IOP interface driver is a list of the addresses of seven subroutines, called the **primary subroutines.** These subroutines function like those of any other Cromix driver that resides on the S-100 bus. The list of primary subroutines for the xyzintf interface driver takes the following form.

```
        entry   xyzintf

xyzintf:dw      init            ; initialize device
        dw      open            ; open device
        dw      close           ; close device
        dw      read            ; read device
        dw      write           ; write to device
        dw      getmode         ; get device mode
        dw      setmode         ; set device mode
```

Each driver is designed to work with a specific device. The device an IOP interface driver works with is the IOP, while it is running certain software. This software is a small operating system, the IOP OS, whose function is to run drivers on the C-bus. The Iopgen utility links the IOP OS into every IOP driver file.

When a system call is made to read the **xyz** device, the Cromix Operating System branches to the read subroutine of **xyzintf.** This subroutine sends a read command to the IOP OS. The **xyz** driver has a list of seven subroutines on the C-bus that correspond to those of the interface driver on the S-100 bus. The IOP OS calls the read command in this list that returns the requested data to IOP OS that in turn delivers it to the read subroutine of **xyzintf.** From there it is returned to the Cromix Operating System in the same way any Cromix S-100 driver does.

Most drivers can use either **iodev** or **iodevb** as its Cromix-to-IOP interface. The read subroutine in **iodev** issues a read command to the IOP OS for each byte wanted from the IOP driver. The read subroutine in **iodevb** receives a block of bytes from the IOP driver in response to one read command to the IOP OS. The write subroutines function in a similar manner. The **iodev** interface is used with terminal and printer drivers that process I/O bytes extensively (expanding tabs and formfeeds, checking for newlines and CONTROL-C signals, processing X-OFF and X-ON characters, etc.), because **iodev** increases the parallel processing of host processor and IOP. The **iodevb** interface is used for drivers such as the tape driver that do not process I/O bytes much, because overhead is reduced by sending fewer commands to the IOP OS.

The choice of the interface (**iodev** or **iodevb**) affects how the corresponding C-bus driver must be written, as will be seen shortly.

The source for the interface drivers is contained in the directory **/cro/driv/iop/host.** The modules of **iodev** are **iodev.z80, iop.z80,** and **iopparm.z80.** The **iodevb** interface uses those modules plus **iodevb.z80** and **iopb.z80.** The same directory also contains the source for **iomem,** the interface to the IOP memory driver, which uses the **iomem.z80, iop.z80,** and **iopparm.z80** modules.

1. **The C-bus Driver**

   A C-bus driver is also a list of the addresses of seven primary driver subroutines, as is an S-100 bus driver. There are a few differences in the operation of C-bus subroutines. These differences result from the fact that they interact with the IOP OS rather than with the Cromix Operating System directly. A description of the operation of C-bus primary driver subroutines follows.

2. **Initialize Subroutine**

   The first subroutine, the initialize subroutine, is called only once, when the IOP drivers are loaded into the IOP. The IOP Qdinit utility serves as the initialization subroutine for drivers that use the Quadart, and the Cspinit utility serves those that use the CSP.

3. **Host-Answering Subroutines**

   Each of the remaining six primary driver subroutines is called by the IOP OS whenever the corresponding subroutine in the interface driver is called by the Cromix Operating System. After one of these six subroutines is called it must respond by calling one of the **host-answering** subroutines provided by the IOP OS unless an error occurs. If an error occurs, the subroutine should not call any host-answering subroutine. Instead it should set the carry flag and return to the IOP OS with the **a** register containing the error number, obtained from the file **/equ/jsysequ.z80.** The IOP OS will return this error number to the host.

The function of the host-answering subroutines is to return data and other information to the interface driver in the host. Refer to the section on Host-Answering subroutines.

All of the following six primary subroutines should indicate error by returning the carry flag set and with the **a** register containing the appropriate error number selected from the file **/equ/jsysequ.z80.** In this case they must not have called any host-answering subroutine. If a subroutine has called any host-answering subroutine, it must itself return with the carry flag reset.

Open Subroutine

This subroutine is called by the IOP OS with the **de** register containing the device number. It should call the Qdsetup IOP utility if it uses the Quadart, and Cspsetup if it uses the CSP. Either of these utilities finds a data structure and an IOP channel for the device. The Qdsetup utility also sets up the interrupts for Quadart I/O, giving the interrupt handler the addresses of the driver's interrupt routines. (The Cspsetup utility does not set up interrupts because the devices that currently use the CSP do not use interrupts.) Unless an error occurs, the open subroutine must return the IOP channel number to the host by calling the Haputc utility and then return with the carry flag reset. The channel number is stored and used by the interface driver in all further communication with the IOP concerning this device.

Close Subroutine

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. If the open subroutine keeps a count of the number of times the device has been opened, the close subroutine should decrement this count each time it is closed. After the open decrement reaches zero, the close subroutine of a driver that uses Quadarts can call the Qdshut utility to relinquish its data area and to mask its interrupts. Unless an error occurs, it should call the Hacontinue utility in order to let the host proceed, and then return with the carry flag reset.

Read Subroutine for Drivers whose Input is Obtained under Control of the Driver

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. If the device input queue has not been appropriately filled, the driver should read the device to fill it. If the corresponding interface driver in the host is **iodevb**, the driver should use the Haputbuf utility to send data to the host a bufferful at a time. If the interface driver is **iodev**, the driver should use the Haputc utility to deliver one byte. In either case it should return with the carry flag reset.

Read Subroutine for Drivers whose Input is Delivered under External Control

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. If the device input queue has not been appropriately filled by the input interrupt subroutine, the read subroutine should call the Hawait utility with the **a** register containing the constant HmRDWAIT and then return with the carry flag reset. This constant, defined in the **waitdef.z80** file, specifies the reason for the wait. If the device has a character available, the read subroutine should retrieve it from the input queue by calling the Getq utility and then deliver it to the host by calling the Haputc utility. It should then return with the carry flag reset.

Input Interrupt Subroutine

Assuming that the open subroutine (see above) has set up interrupts using either the Qdsetup or Addint utility, the interrupt handler will branch to this subroutine when the device hardware causes an input interrupt. This subroutine should read a character or characters from the device and place them into the input queue by calling the Putq utility. If the driver's read subroutine has put the process to sleep by calling the Hawait using the constant HmRDWAIT, then the interrupt subroutine should wake up the process by calling the Hwakeup utility using the same constant after the input queue has been appropriately filled. What constitutes being appropriately filled depends on how the device is to be used. Terminal drivers, for example, usually do not wake up the read process until the queue contains a full line.

Write Subroutine for Drivers whose Output is Transmitted under Control of the Driver

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. If the driver buffer has been filled from earlier calls to the write subroutine, the buffer should now be written to the device. Otherwise, data should be read from the host and placed into the buffer. If the corresponding interface driver in the host is **iodevb**, the write subroutine should use the Hgetbuf utility to get data from the host a bufferful at a time. It the interface driver is **iodev**, the write subroutine should use the Hgetc utility to get one byte. After this is done, it should call the Hacontinue utility to permit the host to proceed and then return with the carry flag reset.

Write Subroutine for Drivers whose Output is Transmitted under External Control

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. It should first call Hgetc to get the byte the host is delivering. If the number of characters already in the output queue exceeds a certain number, say 60, the write subroutine should call the Hawait utility with the **a** register containing the constant HmWRWAIT, and then return with the carry flag reset. Otherwise the write subroutine should call Hacontinue to allow the host to proceed, process the character if appropriate (expand tabs, etc.), place the processed byte or bytes into the output queue by calling the Putq utility, and then return with the carry flag reset.

Output Interrupt Subroutine

Assuming that the open subroutine has set interrupts up using either the Qdsetup or Addint utility, the interrupt handler will branch to this subroutine when the device hardware causes an input interrupt. This subroutine should get a character from the output queue by calling the Getq utility and then write it to the device. If the driver's write subroutine has put the process to sleep by calling the Hawait utility using the constant HmWRWAIT, then the interrupt subroutine should awaken the process by calling the Hwakeup utility with the same constant after the number of characters in the output queue has fallen to a certain number, say 30.

Getmode Subroutine

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. If the corresponding interface driver in the host is either **iodev** or **iodevb**, this getmode subroutine should call the Hgetc utility to get the contents of the user **c** register (the value of the **c** register when the **.getmode** system call was made in the host). To return one byte of data to the user **d** register (the value will be placed in the **d** register when the return from the **.getmode** system call is made in the host), the Haputlast utility should be called. To return a pair of bytes to be placed in the user **de** register, the Haputpair utility should be called. In either case, the getmode subroutine should return with the carry flag reset. On the other hand, if the driver does not use modes, the getmode subroutine should load the **a** register with an appropriate error number (selected from the file **/equ/jsysequ.z80**) and return with the carry flag set.

Setmode Subroutine

This subroutine is called by the IOP OS with both the **de** register and the **iy** register containing the device's data structure address. If the corresponding interface driver in the host is either **iodev** or **iodevb**, this setmode subroutine should call the Hgetc utility to get the contents of the user **c** register. It should then call the Hacontinue utility to tell the host to proceed. The host will next send the contents of the user **de** register, so the setmode subroutine should call Hgetpair to get it. To return one byte of data to the user **d** register, the Haputlast utility should be called. To return a pair of bytes to be placed in the user **de** register, the Haputpair utility should be called. In either case, the setmode subroutine should return with the carry flag reset. On the other hand, if the driver does not use modes, the setmode subroutine should call Hcontinue, call Hgetpair, and then load the **a** register with an appropriate error number (selected from the file **/equ/jsysequ.z80**) and return with the carry flag set.

UTILITIES FOR ADDING AND REMOVING INTERRUPT VECTORS

Most character drivers use interrupts. The following subroutines provide drivers a means of adding interrupt vectors to the system, and of removing them from it.

Note that the Qdsetup utility can also be used to set up interrupts. It uses the Addint utility to do this.

Addint

Call Addint with the Z80 Mode 2 interrupt vector in the **a** register, the address of the interrupt subroutine in the **bc** register, and data which will be loaded into the **de** register whenever the interrupt occurs in the **de** register.

Addint returns with the carry flag set if that interrupt vector has already been used. Otherwise, returns with the carry flag reset.

Addint preserves all registers except the **af** register.

Subint

Call Subint with the Z80 Mode-2 interrupt vector in the **a** register.

Subint preserves all registers except the **af** register.

CHARACTER BUFFERING UTILITIES

A character driver that uses interrupts must buffer the characters it processes unless the device is fast enough to handle the characters at the rate the system can send them. A driver can use the Putq and Getq utilities to buffer characters in a system character queue (first-in, first-out). The space where the characters are stored is provided by the Cromix Operating System. The driver needs to provide five bytes of storage called the queue header which the Putq and Getq utilities will use to keep track of the queue. The queue header consists of a byte count and two pointers. The driver must initialize the byte count to zero. After this, the Putq and Getq utilities handle the byte count and the queue pointers.

An example of a header for a queue named **outq** follows.

```
outq:   ds      1       ; byte count
        ds      2       ; pointer
        ds      2       ; pointer
```

Putq

Call Putq with the address of the queue header in the **hl** register and the character to be put into the queue in the **a** register.

Putq returns with the carry flag set if there was no space to store the character. Otherwise, returns with the carry flag reset.

Putq preserves all registers (including the **af** register).

Getq

Call Getq with the address of the queue header in the **hl** register.

Getq returns with the carry flag set if the queue was empty. Otherwise, returns with the carry flag reset and the character in the **a** register.

Getq preserves all registers except the **af** register.

HOST-ANSWERING SUBROUTINES

The function of the host-answering subroutines is to return data and other information to the interface driver in the host. Among other functions, the host-answering subroutines inform the host that it may proceed to do other things because the command will be executed without further attention from the host. The Hacontinue subroutine, for example, only provides this information. The Haputc subroutine provides this information and also returns a byte of requested data to the interface driver. The host-answering subroutines are Haabort, Hacontinue, Hawait, Haiowait, Haputc, Haputbuf, Haputlast, Haputnot, and Haputpair - the name of each of them begins with the letters **Ha**. A C-bus subroutine should be written so that it calls the appropriate host-answering subroutine as soon as possible, in order to increase the parallel processing of host processor and IOP.

Haabort

This utility is used to answer the host after the host has commanded a sleeping C-bus process to abort. The effect of calling this utility is that the host process will abort itself. See the Haiowait utility below.

Haabort uses no registers for input data and preserves all registers except the **af** register.

Hacontinue

The Hacontinue utility is used when there is no data to return to the host in order to tell the host that its last command has been accepted and will be executed when there is no data to return to the host.

Hacontinue uses no registers for input data and preserves all registers except the **af** register.

Haputc

This utility delivers a byte of data to the host. If the interface driver **iodevb** is being used in the host, the Haputbuf utility should be used to deliver bytes to the host in the C-bus read subroutine rather than this utility.

Call Haputc with the byte in the **a** register.

Haputc preserves all registers except the **af** register.

Haputlast

This utility delivers a byte of data in C-bus read subroutines, telling the host that there are no more. If the interface driver **iodevb** is being used in the host, the Haputbuf utility should be used to deliver bytes to the host in the C-bus read subroutine rather than this utility. It is also used in Getmode and Setmode utilities to return a single byte rather than two. (Some drivers have both one and two byte mode parameters.)

Call Haputlast with the byte in the **a** register.

Haputlast preserves all registers except the **af** register.

Haputpair

This utility is used in the Getmode and Setmode utilities of C-bus drivers to return two bytes of data to the host.

Call Haputpair with the pair of bytes in the **hl** register.

Haputpair preserves all registers except the **af** register.

Haputnot

This utility is called by a read subroutine to report to the host that an end-of-file character has been input. The effect on the host interface driver is to cause it to return without asking for further bytes from the IOP.

Haputnot uses no registers for input data and preserves all registers except the **af** register.

Haputbuf

This utility is called by C-bus read subroutines of drivers which use the **iodevb** interface driver in the host. It sends the contents of a buffer to the host.

Call Haputbuf with the address of the buffer in the **hl** register and the buffer byte count in the **de** register.

Haputbuf alters the **af, de,** and **hl** registers.

Hawait

When a process calls a driver to ask for a resource that the driver does not currently have, it should suspend the process by calling the Hawait utility. When the resource becomes available the Hwakeup utility should be called so that the process can continue execution. The Hwakeup utility is usually called by an interrupt subroutine. For example, a read system call from a driver which has no characters in its input queue should result in a call to the Hawait utility. The Hwakeup utility should be called by the input interrupt subroutine whenever sufficient characters have been placed into the input queue.

Call Hawait with the address of the device data structure in the **iy** register and a constant in the **a** register which specifies the reason for the wait. The following such constants are defined in the file **waitdef.z80.**

HmRDWAIT	Wait until input is ready
HmWRWAIT	Wait until the output queue is sufficiently empty
HmSCWAIT	Wait for a start character (X-ON or CONTROL-Q)
HmWAKIOP	Wait until woken up by the IOP

Hawait returns when the process has been awaken by a call to the Hwakeup utility with the same identification constant.

The Hawait utility preserves all registers except the **af** register.

Haiowait

This utility tells the host process to sleep by calling the Hawait utility with the HmWAKIOP identification constant in the **a** register. It then calls the Iowait utility to put the C-bus driver process to sleep until awaken by the host. When this utility is used, it is expected that an interrupt will wake up the host process by calling the Hwakeup utility with the constant HmWAKIOP. The host will then wake up the C-bus process.

The Haiowait utility uses no registers for input data. It returns with the carry flag set if the sleeping C-bus process was commanded to abort by the host. If this happens the awaken C-bus process must answer the host by calling the Haabort utility and then return with the carry flag set.

Haiowait preserves all registers except the **af** register.

OTHER HOST SUBROUTINES

HGETC

This subroutine returns a byte of data from the host. Drivers which use the **iodevb** interface driver in the host should use the Hgetbuf utility in their write subroutines rather than this utility.

Hgetc preserves all registers except the **a** register.

Hgetbuf

This utility gets data from the host to fill a buffer. It is called by the C-bus write subroutines of drivers which use the **iodevb** interface driver in the host.

Call Hgetbuf with the **hl** register containing the address of the buffer and the **de** register containing the buffer size in bytes.

Hgetbuf returns with the **de** register containing the number of bytes in the buffer remaining unfilled.

Hgetpair

This utility gets a pair of bytes from the host.

Hgetpair returns with the pair in the **bc** register.

Hgetpair alters the **af** and the **bc** registers.

Hwakeup

Call Hwakeup with the same identification constant in the **a** register which was used when the call to the Hawait utility was made.

The Hwakeup preserves all registers except the **af** register.

Httysig

Calling this utility causes the host to be interrupted by the IOP and informed what kind of signal to perform.

Call Httysig with interrupts disabled, the device number in the **hl** register, and the signal type in the **c** register.

Httysig alters the **af** register.

C-BUS PROCESS CONTROL SUBROUTINES

Iowait

This utility suspends the current IOP process. This utility is used after the host process has been suspended by a call to Hawait with the identification constant HmWAKIOP. After the host process is awakened (typically by an interrupt), the host will then wake the suspended IOP process.

Call Iowait with interrupts disabled.

Iowait returns with the carry flag set if the process is aborted.

Iowait alters the **af** register.

Iofork

This utility forks a new C-bus process in order to execute a subroutine.

Call Iofork with interrupts disabled and the address of the subroutine stored at the IOP memory address, Ioforkadr. Passes to the subroutine the contents of all registers except the **f** register, the flags.

Iofork alters the **af** register.

Iotime

This utility suspends the current IOP process for a time.

Call Iotime with interrupts disabled and the **a** register containing the number of tenths of seconds to suspend.

Iotime alters the **af** register.

QUADART UTILITIES

Qdinit

This utility initializes all 16 Quadart serial channels for asynchronous operation by setting the SIO, PIO, and CTC ports. This utility should be specified as the initialization subroutine for all drivers which use the Quadart interface card.

Call Qdinit with interrupts disabled.

Qdinit alters the **af**, **bc**, **de**, and **hl** registers.

Qdssetup

This utility sets up a serial Quadart device driver. Obtain a data area for the device data structure, initialize it, and add the input and output interrupt vectors to the Z80 interrupt page in the IOP.

Call Qdssetup with interrupts disabled, the device number in the **de** register, the address of the input interrupt routine in the **bc** register, and the address of the output interrupt routine in the **hl** register, the address of the external/status interrupt routine (for modems) in the **ix** register, and the address of the C-bus driver (the seven primary subroutines) in the **iy** register.

If no error, finds address of the device data structure (determined by the minor device number), initializes it, sets up the interrupt handler so that it will load the **de** register with the address of the data structure when the interrupt routines are called, and returns with the carry flag reset and with the address of the device data structure in both the **de** and the **iy** registers. If this is the first time the device was opened, the zero-flag is set and the IOP channel number is returned in the **b** register.

If the device number is illegal or if there is no data area available for the device data structure, returns with the carry flag set and an error number in the **a** register.

Qdssetup alters the **af**, **bc**, **de**, **hl**, **ix**, and **iy** registers.

Qdsshut

This utility shuts down a serial Quadart device driver. Remove its interrupts from the Z80 interrupt page and relinquish its data area.

Call Qdsshut with interrupts disabled, the address of the device's data structure in both the **de** and the **iy** registers, and the IOP channel number in the a register.

Qdsshut alters the **af** register.

CSP UTILITIES

Cspinit

This utility initializes all CSP serial channels. This utility should be specified as the initialization subroutine for all drivers which use the CSP interface card.

Call Cspinit with interrupts disabled.

Cspinit alters the **af**, **bc**, **de**, **hl**, and **ix** registers.

Cspsetup

This utility sets up a serial CSP device driver. Obtain a data area for the device data structure and initialize it.

Call Cspsetup with interrupts disabled, the device number in the **de** register, and the address of the C-bus driver (the seven primary subroutines) in the **iy** register.

If no error, finds address of the device data structure (determined by the minor device number) and initializes it, and returns with the carry flag reset and with the address of the device data structure in both the **de** and the **iy** registers. If this is the first time the device was opened, the zero-flag is set and the IOP channel number is returned in the **b** register.

If the device number is illegal or if there is no data area available for the device data structure, returns with the carry flag set and an error number in the **a** register.

Cspsetup alters the **af**, **bc**, **de**, **hl**, and **iy** registers.