Application Note 706
Writing a Device Driver for TINIOS

www.maxim-ic.com

INTRODUCTION

The TINI® platform[1] JVM is powerful enough for very high-level tasks such as serving web pages or remote data logging. However, the Java language only provides limited low-level access necessary for hardware communication. For speed and determinism sake, tasks of this nature are better implemented at an operating system level.

The TINI I/O Subsystem enables communication with devices that require faster communication than the JVM allows. It provides a mechanism for assembly language drivers to act as an intermediary between hardware and the JVM in a standard way. Additional drivers can be installed for communication with new devices. This article examines the TINIOS I/O subsystem, explains how to write a driver as a native library, and provides a simple example in the form of a pipe driver.

GETTING STARTED

Before we begin, you need the TINI SDK version 1.02e or higher. Versions prior to 1.02e did not publicly expose the `com.dalsemi.comm.NativeComm` class that is necessary for driver communication.

This document assumes some familiarity with the TINI Native Interface (TNI) as well as familiarity with 8051 assembly, the `a390` assembler, and the `macro` preprocessor[2].

DRIVER BASICS

A driver in TINIOS is made up of two parts: a set of standard functions to link with TINIOS (the driver interface) and an interrupt or polled routine (the driver) to communicate with the device. Once the interface is registered with TINIOS, it can be accessed using the `com.dalsemi.comm.NativeComm` class.

Accessing a Driver in Java

A device driver is loaded in as a native library. Once loaded, the following `com.dalsemi.comm.NativeComm` methods can be used.

- `public static int open(int port, int stream);`
  Opens a connection to the driver. port represents the driver's port number, which is chosen by the driver when it registers itself. `stream` can be set to `NativeComm.STREAM_STDIN` or `NativeComm.STREAM_STDOUT` (they cannot be logically OR-ed together), depending on if the handle represents an input or output stream. An integer handle is returned, which represents the stream in all other calls. The handle is actually a one-byte unique-identifier value from the I/O subsystem, but promoted to an integer for Java.

- `public static int close(int handle);`
  Closes the stream associated with `handle`. Returns a 1 for success, or a 0 for failure.

[1] Information available at http://www.ibutton.com/TINI.
[2] The TINI SDK, including all tools, are available at ftp://ftp.dalsemi.com/pub/tini.

- `public static int read(int handle, byte [] arr, int timeout, boolean suspend);`
- `public static int read(int handle, byte [] arr, int length, int offset, int timeout, boolean suspend);`

  Reads from the opened stream. `handle` must represent a stream opened for reading. `arr` is the byte array buffer for the input. `length` is the size of the input buffer. `offset` represents the first element of `arr` to be used for the read. `timeout` is the time in milliseconds allowed for the read to complete before it returns. `suspend` should be set to true to ignore the timeout and suspend until the read completes, or false to use the timeout. Function returns the number of bytes read.

- `public static void write(int handle, byte [] arr);`
- `public static void write(int handle, byte [] arr, int offset, int length);`

  Writes to an opened output stream. `handle` must represent a stream opened for writing. `arr` is the output array. `offset` is the first element of `arr` to be written. `length` is the number of bytes to write.

- `public static int ioctl(int handle, byte [] arr, int offset, int length, int timeout);`

  Sends an ioctl call to the driver. `handle` is any valid stream handle. `arr` is an array of parameters. `length` is the number of parameters. All other arguments should be ignored.

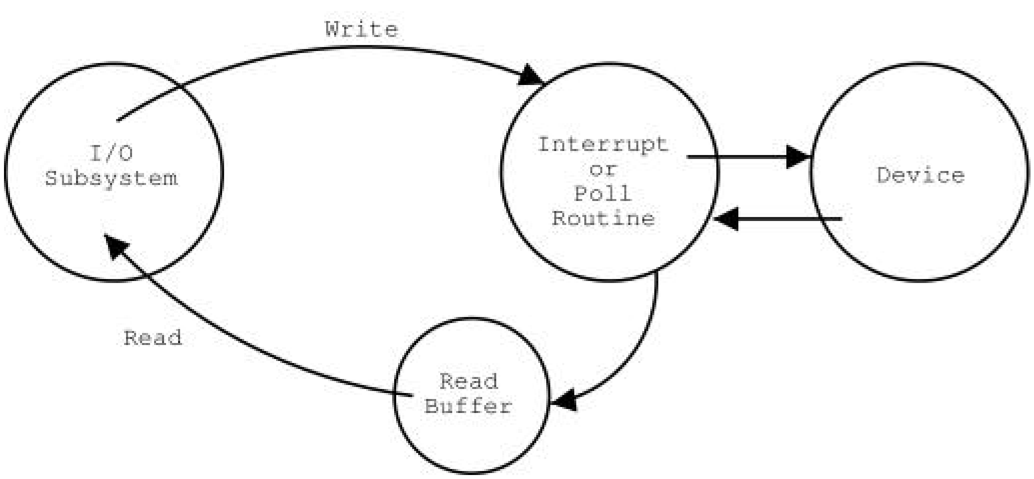
- `public static int available(int handle);`

  Returns the number of bytes available without blocking. `handle` represents an input stream.

DRIVER PHILOSOPHY

Figure 1 shows the design of a TINIOS device driver.

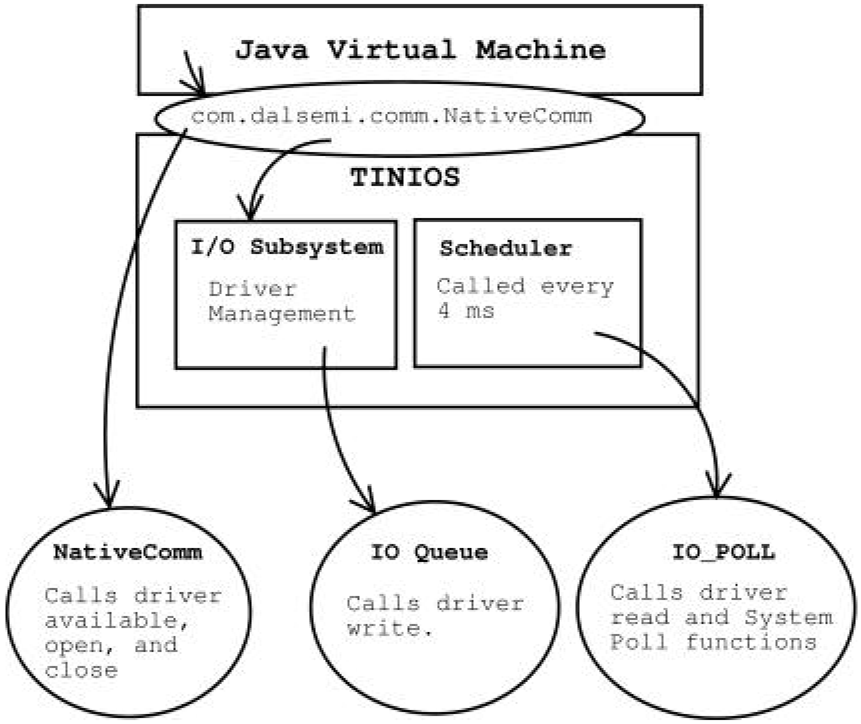
Figure 1. I/O DRIVER PHILOSOPHY



The driver can store incoming data in some sort of buffer in the event a read cannot be handled immediately. A write is directly fed from the driver interface to the driver.

The driver interface consists of implementations of open, close, read, write, available, and ioctl functions. These can be grouped together by what portion of the I/O subsystem invokes them, as shown by Figure 2.

Figure 2. TINIOS I/O DESIGN



NativeComm

The `com.dalsemi.comm.NativeComm` class binds the Java Virtual Machine and the I/O subsystem. It contains static Java methods for calling the driver interface functions. While the `read` and `write` driver calls are queued by the I/O subsystem, the `open`, `close`, `available`, and `ioctl` are invoked directly from the NativeComm methods. Program execution continues uninterrupted after these functions complete.

- `open`: Called when the user tries to open a device for input or output. Driver and device initialization should be performed in this function. Returns a success/error value.
- `close`: Called when the user closes a device. Driver and device shutdown procedures should be performed in this function. Returns a success/error value.
- `available`: Called to check the amount of available data that can be read from the driver. Returns an integer count.
- `ioctl`: Called for functionality not directly supported by the driver interface. Returns a success/error value.

I/O Queue

The I/O subsystem handles writes differently than the above methods.

1) The I/O subsystem receives the `write` call, checks if the handle is open for writing, and queues it if the device is busy.
2) The driver `write` function is passed a buffer and a byte count to write. The driver `write` is a catalyst—it does not perform the write operation, but instead prepares the driver for writing and returns.
3) The I/O subsystem puts the executing Java thread to sleep.
4) The driver performs the hardware write. When it is finished, it calls the `IO_WriteFree` function to inform the I/O Subsystem it has completed.
5) The Java thread is awoken, and execution resumes.

It is important to note that the driver `write` does not write to the hardware, but only prepares the write for asynchronous execution by the driver. The safest approach is to have a poll routine check for the completion of the write, calling `IO_WriteFree` when it detects completion. If `IO_WriteFree` is called from the `write` function, it attempts to wake the Java Thread before it is put to sleep, and the **thread suspends indefinitely**. Also, TINI does not have any way of protecting operating system state from being modified inside of an interrupt service routine. Therefore, `IO_WriteFree` should **not** be called from an ISR.

IO_POLL

The I/O subsystem must check on registered drivers continuously. To perform these checks, the IO_POLL function is called by the scheduler every 4ms[3]. IO_POLL manages the status of pending reads and writes. It is possible to register your own callback functions to be called on every IO_POLL. These do not necessarily need to be I/O related and can therefore serve any purpose.

The `read` driver interface function is called by IO_POLL as well. A driver `read` behaves very differently than a driver `write`.

1) The read is queued by the I/O subsystem.
2) The Java thread is put to sleep.
3) IO_POLL calls the driver `read` function. The driver read uses an intermediate buffer of its own to return up to 250 bytes back to the I/O subsystem. The driver `read` copies the amount it has available into the buffer and exits.
4) Step 3 repeats until the amount requested is read in, or a timeout occurs in IO_POLL (if a timeout is enabled).
5) The Java thread is awoken by the I/O subsystem upon completion of the read or a timeout.

Note the differences between write and read. While a driver `write` is called once to initiate a write to a device, a driver `read` is called repeatedly by the I/O subsystem to poll the data. Also, while a write operation requires an `IO_WriteFree` for termination, a read operation is terminated automatically by the I/O subsystem on completion.

[3] More or less 4ms. TINIOS is not real-time.

IMPLEMENTATION DETAILS

Driver Portion
If the device uses an interrupt, use `System_Install_Interrupt` to install your interrupt service routine, or `System_Register_External_Interrupt` to chain to the external interrupt line. If the device requires a poll-based routine, use `System_RegisterPoll` to have your routine called every 4ms by IO_POLL.

Device writes should be handled within the driver, calling `IO_WriteFree` on completion. It is important to note that `IO_WriteFree` cannot be called by the driver write interface function or by any interrupt service routine.

Driver Interface Registration
To add a driver to the TINIOS, you need to register your driver with the I/O subsystem using the function `System_Register_Driver`. A driver number between IOSYS_USER_DRIVER and IOSYS_MAX_DRIVER must be assigned to the driver[4]. This number is used when opening and closing the port.

It is recommended that you perform this step in the initialization function of your native library. The driver will not be available until this step is complete.

Open
The driver open function is called when a thread attempts to open a driver. One of the microcontroller registers (R2_B0) is set to the handle assigned to the port. Any state and device initialization should be performed here.

Clear the accumulator to denote success, or an exception number (defined in `apiequ.inc`) to throw an exception. All other registers must be restored to their prior state before returning.

Close
The driver close function is called when a thread closes the driver. Resident state cleanup and device shutdown should be handled here.

Clear the accumulator to denote success, or an exception number (defined in `apiequ.inc`) to throw an exception. All other registers are restored to their prior state before returning.

Read
Once a device read is initiated, the driver read function is called repeatedly. It is not passed any arguments, and it returns a pointer to a structure, described by Figure 3.

Figure 3. READ FUNCTION RETURN VALUE

[4] TINIOS supports 16 I/O drivers. Drivers 0 to 7 are allocated to the system, and 8 to 15 are allocated to the user.

The first byte is the size of the buffer being returned, with a maximum or 250 bytes. The next three bytes are reserved, and should be set to 0. The remaining bytes are the data from the resident portion's read buffer. *Do not create a new buffer on every read call!* Instead, create a buffer in the `open` function and free it in `close`. If speed is not an issue, you can use ephemeral state blocks (with `NatLib_GetEphemeralStateBlock` and `NatLib_RemoveEphemeralStateBlock`) to hold your state information. Otherwise, use indirects (acquired from `System_AcquireIndirectSemaphore`) to store the pointer to your state.

Clear the accumulator to denote success, or an exception number (defined in `apiequ.inc`) to throw an exception. All registers except the first data pointer must be restored before returning.

Write

The driver write is passed the application id, the thread id, the data length, and a pointer to the data to be written. All of this information must be passed to the resident portion and the `write` must be initiated.

Clear the accumulator to denote success, or an exception number (defined in `apiequ.inc`) to throw an exception. All other registers must be restored on completion.

Available

Available is called to query the amount of data that can be read from the driver without blocking. The value can be returned in R3:R2:R1:R0 of register bank 1. Clear the accumulator to denote success, or an exception number (defined in `apiequ.inc`) to throw an exception. All other registers are restored to their prior state before returning.

ioctl

The ioctl (input output control) function is used to make device specific calls. The first data pointer point to an argument array, whose length is stored in R5:R4.

Clear the accumulator to denote success, or an exception number (defined in `apiequ.inc`) to throw an exception. A return value can be put in R3:R2:R1:R0 of register bank 1. All other registers must be restored on completion.

Process Destruction

A driver can register a process destroy callback, using `System_RegisterProcessDestroyFunction`, to perform any closing and shutdown operation when a process exits. The process destroy callback is passed the identifier of the dying process in R0. Clear the accumulator to denote success, or an exception number to throw an exception. All other registers must be restored on completion.

Example: System Pipe

Provided in the appendices is a sample driver that implements a very simple pipe driver. The pipe driver allows for output from one process or thread to be read from another process or thread. It uses an Ephemeral State Block (ESB) to hold the circular buffer and the read function return buffer. One byte is used for the size count, and another is used for the offset of the first byte into the queue. Using a 256-byte buffer makes handling the rollover simple, as it occurs automatically when the appropriate registers overflow.

On startup, the native library initialization routine uses the process id from System_GetCurrentProcessId (which is assigned a value of 0 to 7) to set a bit in the reference counter value of the ESB. It then registers a process destroy callback, which clears the bits of the reference counter. When the counter is 0, the driver performs cleanup operations and unregisters itself. This allows communication across process boundaries.

Memory past the circular buffer is used for the read return value. There is no need to worry about serializability of the return buffer, because only one driver `read` can be called at a time. The read function insures it copies less than 250 bytes before performing the copy.

The write function installs a poll routine in IO_POLL to write the data into the buffer as space becomes available. The poll routine performs the copy, storing intermediate state near the end of the ESB. The poll routine removes itself when the write completes.

Classes in the `com.dalsemi.pipeio` package implement a `java.io.InputStream` and `java.io.OuputStream` on top of the driver. This allows for java programs to access the driver as if it was another standard Java input or output stream.

CONCLUSION

While most embedded tasks can be handled by the TINI JVM, there are instances that require more greater and tighter constraints. These can be handled by adding native drivers to TINIOS. A native driver consists of a polled or interrupt handler, and a set of interface functions. They can be loaded inside of a native library, and TINI provides a standard interface to communicate with them. The main advantage of using a native driver is that most of the operating system tasks, like queuing multiple writes or blocking a process on a read, are offloaded onto TINIOS. This frees the developer to focus on device communication and not operating system semantics.

TINI is a registered trademark of Dallas Semiconductor.

Appendix A: driver.a51

```
$include(ds80c390.inc)
$include(apiequ.inc)
$include(tini.inc)
$include(tinimacro.inc)
$include(driver.inc)

;
; This driver implements a 255 byte circular buffer for
; communication across threads or processes. It user
; an ephemeral state block to hold the circular buffer
; and the return value for the read. (The structure
; is described in driver.inc).
;


;****************************************************************************
;*
;* Function Name: Init_Pipe
;*
;*   Description: Native Library initialization routie.
;*
;*      Input(s): None.
;*
;*    Outputs(s): a - 0 on success, non-zero for init exception.
;*
;*         Notes: I am creating my state info in this function.
;*
;*
;****************************************************************************


Init_Pipe:
        ;
        ; This function performs two operations. The first is
        ; to register the driver with the I/O subsystem. The second
        ; is to create state info for the pipe (the circular buffer,
        ; the read return value buffer, and the write state
        ; buffer).
        ;

Init_GetEphemeralStateBlock:
        ;
        ; Now create the state info. First, check if the
        ; Ephemeral State Block (ESB) exists.
        ;
        mov   dptr, #QueueID           ; see if we have an ESB installed
        lcall NatLib_GetEphemeralStateBlock


        ;
        ; Cache pointer away if it gets destroyed
        ;
        mov     R5,dpl
        mov     R6,dph
        mov     R7,dpx

        jnz   Init_MallocEphemeral     ;
```

```
        ;
        ; an ESB exists - some process
        ; has already called load since
        ; boot time
        ;
Init_StoreRetrieveEphemeral:
        ljmp  Init_IncrementReferenceCount

        ;
        ; No ESB exists! Dios Mio! Time to
        ; init all the data!
        ;

Init_MallocEphemeral:
        ;
        ; Install the process destroy function
        ;
        mov     dptr, #Pipe_ProcessExit
        lcall   System_RegisterProcessDestroyFunction

        mov   R2, #LOW(PIPE_END)        ; Allocate 1024 bytes
        mov   R3, #HIGH(PIPE_END)        ;
        lcall mm_Malloc                 ;
        jnz   Init_Done                 ;

        ;
        ; The memory is cleared by mm_malloc, so there isn't
        ; any more initialization to do
        ;
        mov   dpl1, dpl                 ; copy address to
        mov   dph1, dph                 ;  second dptr
        mov   dpx1, dpx                 ; handle is already in R3:R2

        ;
        ; Cache pointer away if it gets destroyed
        ;
        mov     R5,dpl
        mov     R6,dph
        mov     R7,dpx

        mov   dptr, #QueueID            ; first dptr is our identifier
        lcall NatLib_InstallEphemeralStateBlock

        mov     dpl,R5
        mov     dph,R6
        mov     dpx,R7

        ;
        ; We need to now assign our driver a
        ; driver number and store it away
        ;
        mov     a,#LOW(PIPE_DRVNUM)
        mov     b,#HIGH(PIPE_DRVNUM)
        lcall   add_dptr1_16
        lcall   System_IO_NextAvailableDriverNum
        PUTX
        mov     dpl,R5
        mov     dph,R6
        mov     dpx,R7

        ;
```

```
        ; Register the driver with the I/O subsystem.
        ; acc holds the driver num
        ;
        lcall   Pipe_Init
        push    acc
        lcall   System_Register_Driver
        ;
        ; Disable non blocking write I/O
        ;
        pop     acc
        mov     b,#01
        lcall   System_IO_EnableNonBlockingWrites

        mov     dpl,R5
        mov     dph,R6
        mov     dpx,R7


Init_IncrementReferenceCount:
        ;
        ; mm_malloc should initialize the memory to zero.
        ; but we need to increment the reference count
        ;
        mov     a,#LOW(PIPE_REFCOUNT)
        mov     b,#HIGH(PIPE_REFCOUNT)
        lcall   add_dptr1_16
        ;
        ; Set the bit denoting the process id
        ;
        lcall   System_GetCurrentProcessId
        lcall   Pipe_Power
        mov     b,a
        GETX
        orl     a,b
        PUTX


Init_Done:
        clr     a
        ret

;****************************************************************************
;*
;* Function Name: Pipe_Read
;*
;*   Description: Function to copy data from the circular buffer
;*                back to the I/O subsystem.
;*
;*      Input(s): None
;
;*    Outputs(s): dptr  - pointer to input buffer
;*
;*         Notes: First 4 bytes of read buffer are:
;*                           length, 0, 0, 0
;*
;****************************************************************************

        ;
        ; The function copies data out of the circular buffer
        ; and into the read return buffer. I/O allows a maximum
        ; return value of 250 bytes per read, so that is our upper
```

```
        ; bound
        ;

Pipe_Read:

        ;
        ; First, save *everything* away.
        ;
        PUSH_DPTR2
        PUSH_BANK_0
        PUSH_BANK_1
        PUSH b
        PUSH acc
        PUSH DPS

        ;
        ; Get the state block
        ;
        PIPE_GET_BUFFER
        ;
        ; Save the pointer away
        ;
        mov     R0_B1,dpl
        mov     R1_B1,dph
        mov     R2_B1,dpx
        PUSH_DPTR1

        ;
        ; Move to the read buffer
        ;
        mov     a,#LOW(PIPE_R_LEN)
        mov     b,#HIGH(PIPE_R_LEN)
        lcall  add_dptr1_16
        ;
        ; Store the read buffer away in DPL1:DPH1:DPX1
        ;
        mov     R3_B1,dpl
        mov     R4_B1,dph
        mov     R5_B1,dpx

        mov     dpl1,dpl
        mov     dph1,dph
        mov     dpx1,dpx
        POP_DPTR1
        ;
        ; Now DPL:DPH:DPX is pointing to the pipe buffer
        ; and DPL1:DPH1:DPX1 is pointing to the read buffer.
        ; Get the state vars from the pipe
        ;

        mov     dps,#0                          ; Use DPL:DPH:DPX
        inc     dptr                            ; Move to the size byte
        GETX
        mov     R0,a                            ; Get the start offset
        inc     dptr
        GETX
        mov     R1,a
        inc     dptr

        ;
        ; R7 is R0 with the high bit cleared. It will
```

```
        ; serve as our loop counter and as our number
        ; of bytes to read
        ;
        mov     R7,R0_B0
        clr     c
        mov     a,R0
        cjne    a,#250,$+3
        jc      pipe_read_init_ret_buf
        mov     R7,#250

pipe_read_init_ret_buf:
        ;
        ; We're going to need this buffer later
        ;
        PUSH_DPTR1

        ;
        ; Ok, now the size and offset are in R0:R1.
        ; and DPL:DPH:DPX are pointing to the first byte
        ; of the pipe. Lets initialize the return buffer,
        ; shall we?
        ;
        ; Start by switching to DPL1:DPH1:DPX1
        ;
        inc     dps
        ;
        ; Write the length of the return size
        ;
        mov     a,R7
        PUTX
        inc     dptr
        ;
        ; Fill in the rest of the header
        ;
        clr     a
        PUTX
        inc     dptr
        PUTX
        inc     dptr
        PUTX
        inc     dptr

        ;
        ; We're going to copy what we can from the pipe
        ; into the read buffer.
        ;

        POP_DPTR1



        ;
        ; Looks like we're committed. Lets start reading
        ;
        mov     dps,#1

pipe_read_test_zero:
        ;
        ; Test if we are sending zero bytes
        ;
        mov     a,R7
        jnz     pipe_read_loop
```

```
        ljmp    pipe_read_exit

pipe_read_loop:
        ;
        ; Use DPL:DPH:DPX
        ;
        inc     dps
        ;
        ; Use movc for the copy
        ;
        ; Yes, I *know* how very illegal this should be
        ; as it is only possible when code space = data space,
        ; but I saw Don "The Godfather" Loomis ("Let me write you
        ; a firmware you can't refuse") do it in his code, so
        ; I claim I can to.
        ;
        mov     a,R1
        movc    a,@a+dptr


        ;
        ; Write acc into the read buffer
        ;
        inc     dps
        PUTX
        inc     dptr

        ;
        ; increment the buffer offset, decrement
        ; the size count, and loop
        ;
        inc     R1
        dec     R0
        djnz    R7,pipe_read_loop


pipe_read_exit:
        ;
        ; Get the read buffer
        ;
        mov     dpl, R0_B1
        mov     dph, R1_B1
        mov     dpx, R2_B1

        ;
        ; Update 'start' and 'offset'
        mov     dps,#0
        inc     dptr
        mov     a,R0
        PUTX
        inc     dptr
        mov     a,R1
        PUTX


        mov     dpl, R3_B1
        mov     dph, R4_B1
        mov     dpx, R5_B1
```

```
        ;
        ; Exit without incident
        ;

        POP DPS
        POP acc
        POP b
        POP_BANK_1
        POP_BANK_0
        POP_DPTR2

        clr     a
        ret

;****************************************************************************
;*
;* Function Name: Pipe_Write
;*
;*   Description:
;*
;*      Input(s): dptr  -> pointer to data to send
;*                R2    -> App Id
;*                R7    -> Thread Id
;*                R5:R4 -> length of data to send.
;*
;*    Outputs(s): a - 0 on success, exception number on failure
;*
;*
;****************************************************************************

        ;
        ; This function "initiates" the write operation.
        ; It moves the state it has been passed into the ESB,
        ; and then installs a poll routine to perfom the copy
        ; operation. The function then exits.
        ;


Pipe_Write:
        ;
        ; Get the state block, but make sure not
        ; to disrupt any of the registers we've
        ; been passed!
        ;
        PUSH acc
        PUSH b
        PUSH dps
        PUSH_BANK_0
        PUSH_DPTR1


        push    r4_b0
        push    r5_b0
        push    r2_b0
        push    r7_b0
        PIPE_GET_BUFFER

        pop     R7_b0
        pop     R2_B0
        pop     R5_B0
        pop     R4_B0
```

```
mov     dps,#0


;
; Write all of our state while making sure
; the scheduler doesn't run (preventing
; our poll from writing in the middle of
; our write).
;

TINIOS_ENTER_CRITICAL_SECTION

;
; Inform our poll routine that a write is occurring
;
GETX
setb    PIPE_WRITE_BIT
PUTX

;
; Fill in the state vars
;
mov     a, #LOW(PIPE_W_STATE)
mov     b, #HIGH(PIPE_W_STATE)
lcall   add_dptr1_16



mov     a,R4
PUTX
inc     dptr
mov     a,R5
PUTX
inc     dptr
mov     a,R7
PUTX
inc     dptr
mov     a,R2
PUTX

POP_DPTR2

inc     dptr
mov     a,dpl1
PUTX
inc     dptr
mov     a,dph1
PUTX
inc     dptr
mov     a,dpx1
PUTX
clr     a

mov     dptr,#Pipe_Poll
lcall   System_RegisterPoll

TINIOS_EXIT_CRITICAL_SECTION

POP_BANK_0
pop dps
```

```
        pop b
        pop acc
        ret


;*****************************************************************************
;*
;* Function Name: Pipe_Poll
;*
;*   Description: Poll routine installed to copy from the write
;*                buffer into the circular buffer.
;*
;*      Input(s): None
;*
;*    Outputs(s): a - 0 on success, exception number on failure
;*
;*
;*****************************************************************************

        ;
        ; This function performs the write operation. If there is not
        ; enough space in the circular buffer for the data, it will
        ; copy data into the space available, and then return. This
        ; will repeat until all the data has been written, where
        ; the poll routine calls IO_WriteFree and removes itself
        ; from IO_POLL
        ;

Pipe_Poll:
        PUSH_DPTR1
        PUSH_DPTR2
        PUSH_BANK_0
        PUSH_BANK_1
        PUSH b
        PUSH acc
        PUSH DPS

        ; Register Usage
        ; ----------------
        ; R0_B0 - Buffer size
        ; R1_B0 - Offset into Queue
        ; R2_B0 - Handle for the io subsystem
        ; R3_B0 - Count register for loop
        ; R4:R5 - Size for write (Bank 0)
        ; R7_B0 - Thread ID of write
        ; R0:R1:R2 (Bank 1) - Pointer to circular buffer offset
        ; R3:R4:R5 (Bank 1) - Pipe Data Structure

        ;
        ; If the write bit is not set, then
        ; exit out
        ;
        mov     dps,#0

        PIPE_GET_BUFFER
        GETX
        jb      PIPE_WRITE_BIT, pipe_poll_init
        clr     a
        ljmp    pipe_poll_exit

pipe_poll_init:
```

```
        ;
; Cache the pointer to the state block
;
mov     R3_B1,dpl
mov     R4_B1,dph
mov     R5_B1,dpx


mov     a,#LOW(PIPE_W_STATE)
mov     b,#HIGH(PIPE_W_STATE)
lcall   add_dptr1_16
GETX
mov     R4,a
inc     dptr
GETX
mov     R5,a
inc     dptr
GETX
mov     R7,a
inc     dptr
GETX
mov     R2,a
inc     dptr
GETX
push    acc
inc     dptr
GETX
push    acc
inc     dptr
GETX
mov     dpx,a
pop     dph
pop     dpl


;
; Hold onto the write buffer
;
mov     dpl1,dpl
mov     dph1,dph
mov     dpx1,dpx

;
; Cache the pointer to the state block
;
mov     dpl,R3_B1
mov     dph,R4_B1
mov     dpx,R5_B1

;
; Get the size and the offset
;
inc     dptr
GETX
mov     R0,a
inc     dptr
GETX
mov     R1,a
inc     dptr
;
; Cache the pointer to the buffer
```

```
        ;
        mov     R0_B1,dpl
        mov     R1_B1,dph
        mov     R2_B1,dpx
        ;
        ; Calculate the available size for the circular
        ; buffer (255-size = compliment of R0) and stash
        ; in R3
        ;
        mov     a,R0
        cpl     a
        mov     R3,a


        ;
        ; Figure out how many bytes will
        ; be copied in. My logic is:
        ; 1) If R5 is set, then size > 255. Copy in
        ;    what size is available
        ; 2) If R5 is zero and R4 > available size,
        ;    then copy in available size bytes
        ; 3) Otherwise, copy in R4 number of bytes
        ;
pipe_poll_calc_available:
        mov     a,R5
        jnz     pipe_poll_use_available

        clr     c
        mov     a,R3
        cjne    a,R4_B0,$+3
        jc      pipe_poll_use_available

        mov     R3,R4_B0

pipe_poll_use_available:
        ;
        ; Check if R3 is zero, and abort if it is
        ;
        mov     a,R3
        jnz     pipe_poll_not_zero
        sjmp    pipe_poll_notdone



pipe_poll_not_zero:

        ;
        ; Since we aren't going to overwrite the start
        ; variable, lets use it as our offset var into
        ; the circular buffer. Add the size to the offset
        ; to calculate start position
        ;
        mov     a,R0
        add     a,R1
        mov     R1,a

        ;
        ; Subtract the size off R4:R5
        ;
        clr     c
        mov     a,R4
```

```
        subb    a,R3
        mov     R4,a
        mov     a,R5
        subb    a,#0
        mov     R5,a


pipe_poll_loop:
        ;
        ; Yes, I *know* this is a slow approach, but
        ; it means I don't need to do any modulo work.
        ; Get a byte from the write buffer
        ;
        inc     dps
        GETX
        mov     b,a
        inc     dptr

        ;
        ; Return to the circular buffer
        ;
        inc     dps

        ;
        ; Reset it, and go in offset ammount, and write
        ; the byte
        ;
        mov     dpl,R0_B1
        mov     dph,R1_B1
        mov     dpx,R2_B1
        mov     a,R1
        lcall   add_dptr1
        mov     a,b
        PUTX

        ;
        ; Increment counters and loop
        ;
        inc     R1
        inc     R0
        djnz    R3,pipe_poll_loop



        ;
        ; If there are not any other
        ; bytes to write then clear the write
        ;
        mov     a,R4
        orl     a,R5
        jnz     pipe_poll_notdone


        mov     dpl,R3_B1
        mov     dph,R4_B1
        mov     dpx,R5_B1
        GETX
        clr     PIPE_WRITE_BIT
        PUTX

        lcall   System_IO_WriteFree
```

```
        mov     dptr,#Pipe_Poll
        lcall   System_RemovePoll



pipe_poll_notdone:
        ;
        ; Write the current state of the queue
        ; back into the state block
        ;

        mov     dpl,R3_B1
        mov     dph,R4_B1
        mov     dpx,R5_B1
        inc     dptr
        mov     a,R0
        PUTX

        mov     a,#LOW(PIPE_W_STATE-PIPE_SIZE)
        mov     b,#HIGH(PIPE_W_STATE-PIPE_SIZE)
        lcall   add_dptr1_16

        mov     a,R4
        PUTX
        inc     dptr
        mov     a,R5
        PUTX
        inc     dptr
        mov     a,R7
        PUTX
        inc     dptr
        mov     a,R2
        PUTX
        inc     dptr
        mov     a,dpl1
        PUTX
        inc     dptr
        mov     a,dph1
        PUTX
        inc     dptr
        mov     a,dpx1
        PUTX

        clr     a

pipe_poll_exit:


        POP DPS
        POP acc
        POP b
        POP_BANK_1
        POP_BANK_0
        POP_DPTR2
        POP_DPTR1

        clr     a
        ret

;*****************************************************************************
;*
```

```
;* Function Name: Pipe_Open
;*
;*   Description: Function to initialize the driver
;*
;*      Input(s): a     -> Port Number
;*                R2    -> Driver Handle
;*
;*    Outputs(s): a - 0 if success, exception number otherwise
;*
;*         Notes: I am using the Native Library init routine to
;*                create the state block, so we will just initialize
;*                it in this function.
;*
;****************************************************************************

Pipe_Open:
        ;
        ; Initialize the pipe data structure
        ;
        PIPE_GET_BUFFER
        clr     a
        PUTX
        inc     dptr
        PUTX
        inc     dptr
        PUTX

        ;
        ; Clear out anything that remains in the pipe read buffer
        ;
        mov     a,#LOW(PIPE_R_LEN-PIPE_START)
        mov     b,#HIGH(PIPE_R_LEN-PIPE_START)
        lcall   add_dptr1_16
        clr     a
        PUTX
        inc     dptr
        PUTX
        inc     dptr
        PUTX
        inc     dptr
        PUTX
        inc     dptr

        ;
        ; Increment the reference count
        ;

        clr     a
        ret

;****************************************************************************
;*
;* Function Name: Pipe_Close
;*
;*   Description: Function to shutdown the driver
;*
;*      Input(s): a     -> Port Number
;*
;*    Outputs(s): a - 0 if success, exception number otherwise
;*
;*         Notes: Simply making sure the Poll routine has been removed
```

```
;*                  before exiting.
;*
;*
;****************************************************************************


Pipe_Close:
        ;
        ; First, remove the poll routine if it is resident
        ;
        mov     dptr,#Pipe_Poll
        lcall   System_RemovePoll


        clr     a
        ret

;****************************************************************************
;*
;* Function Name: Pipe_Ioctl
;*
;*   Description: Use to make driver-specific calls
;*
;*      Input(s): dptr  -> pointer to argument array
;*                R2    -> Driver Handle
;*                R5:R4 -> length of data to send.
;*
;*    Outputs(s): a - 0 if success, exception number otherwise
;*
;*         Notes:
;*
;****************************************************************************



Pipe_IOCTL:
        clr     a
        ret


;****************************************************************************
;*
;* Function Name: Pipe_Available
;*
;*   Description: Use to make driver-specific calls
;*
;*      Input(s): R2    -> Driver Handle
;*
;*    Outputs(s): a - 0 if success, exception number otherwise
;*                R0:R1:R2:R2 - Size of available pipe data.
;*
;*         Notes:
;*
;****************************************************************************

        ;
        ; Pulls the size out of the ESB and returns it
        ;

Pipe_Available:
        PUSH_DPTR1
```

```
        PUSH_BANK_0
        PUSH acc
        PUSH b

        PIPE_GET_BUFFER


        TINIOS_ENTER_CRITICAL_SECTION
        inc     dptr
        GETX
        TINIOS_EXIT_CRITICAL_SECTION

        mov     R0_B1,a
        clr     a
        mov     R1_B1,a
        mov     R2_B1,a
        mov     R3_B1,a

        POP b
        POP acc
        POP_BANK_0
        POP_DPTR1
        ret

;*****************************************************************************
;*
;* Function Name: Pipe_Init
;*
;*   Description: Prepares the driver for registration
;*
;*      Input(s): None
;*
;*    Outputs(s): All registration registers are set.
;*
;*
;*****************************************************************************


Pipe_Init:
        mov dptr,#Pipe_Read
        mov R0_B0,dpl
        mov R1_B0,dph
        mov R0_B1,dpx

        mov dptr,#Pipe_Write
        mov R2_B0,dpl
        mov R3_B0,dph
        mov R2_B1,dpx

        mov dptr,#Pipe_Open
        mov R4_B0,dpl
        mov R5_B0,dph
        mov R4_B1,dpx

        mov dptr,#Pipe_Close
        mov R6_B0,dpl
        mov R7_B0,dph
        mov R6_B1,dpx

        mov dptr,#Pipe_IOCtl
```

```
        mov R0_B2,dpl
        mov R1_B2,dph
        mov R0_B3,dpx

        mov dptr,#Pipe_Available
        mov R2_B2,dpl
        mov R3_B2,dph
        mov R2_B3,dpx
        ret

;*****************************************************************************
;*
;* Function Name: Pipe_Power
;*
;*   Description: Helper method - performs a 2^a operation
;*
;*      Input(s): acc - value
;*
;*    Outputs(s): acc - Two to the power of parameter (8 bits only)
;*
;*
;*****************************************************************************


Pipe_Power:
        push    b
        mov     b,a
        mov     a,#1
        jz      pipe_power_exit
pipe_power_loop:
        rl      a
        djnz    b,pipe_power_loop

pipe_power_exit:
        pop     b
        ret

;*****************************************************************************
;*
;* Function Name: Pipe_ProcessExit
;*
;*   Description: Called on process destruction. Cleans up
;*                things left behind.
;*
;*      Input(s): acc - Process ID
;*
;*    Outputs(s): none
;*
;*       Note(s): PID is passed in R0
;*
;*****************************************************************************

Pipe_ProcessExit:
        PUSH_DPTR1
        PUSH_BANK_0
        push    b
        push    R0_B1
        push    R1_B1
        push    R2_B1

        PIPE_GET_BUFFER
```

```
        jz      pipe_processexit_start
        ljmp    pipe_processexit_exit

pipe_processexit_start:
        ;
        ; Hold onto the pointer
        ;
        mov     R0_B1,dpl
        mov     R1_B1,dph
        mov     R2_B1,dpx
        ;
        ; Hold onto the handle
        ;
        mov     R6,R2_B0
        mov     R7,R3_B0
        ;
        ; Hold the PID
        ;
        ;
        ; We need to clear the process bit
        ;
        mov     a,#LOW(PIPE_REFCOUNT)
        mov     b,#HIGH(PIPE_REFCOUNT)
        lcall   add_dptr1_16
        ;
        ; Set the bit denoting the process id
        ;
        mov     a,R0
        lcall   Pipe_Power

        cpl     a
        mov     b,a
        GETX
        anl     a,b
        PUTX


        ;
        ; If no one else is holding onto this driver,
        ; then perform the proper cleanup.
        ;
        jz      pipe_processexit_cleanup
        sjmp    pipe_processexit_exit

pipe_processexit_cleanup:
        ;
        ; Since we're still in the neighborhood,
        ; lets get the driver num and store it away
        ;
        inc     dptr
        GETX
        ;
        ; First, remove the driver from the driver table
        ;
        lcall   System_Unregister_Driver


        ;
        ; Next, remove the poll routine if it is resident
        ;
        mov     dptr,#Pipe_Poll
```

```
        lcall   System_RemovePoll

        ;
        ; Remove ourselves from the process destroy
        ; callback
        ;
        mov     dptr, #Pipe_ProcessExit
        lcall   System_UnregisterProcessDestroyFunction

        ;
        ; Finally, lets clear the ESB
        ;
        mov     dptr,#QueueID
        lcall   NatLib_RemoveEphemeralStateBlock

        mov     R2,R6_B0
        mov     R3,R7_B0
        lcall   MM_Free
pipe_processexit_exit:
        pop     R2_B1
        pop     R1_B1
        pop     R0_B1
        pop     b
        POP_BANK_0
        POP_DPTR1
        clr     a
        ret


;
; TINIConvertor in version 1.02 requires a native method
;
Native_GetPipeDriver:
        PIPE_GET_BUFFER
        mov     a,#LOW(PIPE_DRVNUM)
        mov     b,#HIGH(PIPE_DRVNUM)
        lcall   add_dptr1_16
        GETX
        mov     R0,a
        clr     a
        mov     R1,a
        mov     R2,a
        mov     R3,a
        ret


QueueID:
db "DSPIPEEX",0

Debug_Str:
db "Debug Point ",0



END
```

Appendix B: driver.inc

```
$include(tinidriver.inc)

;
;
; Structure for my buffer
;
; struct PipeQueue
; {
;     //
;     //  Circular buffer state vars
;     //
;     u1  flags
;     u1  size
;     u1  start
;     u1  buffer[256]
;     //
;     // Read return value
;     //
;     u1  r_len
;     u1  r_reserved[3]
;     u1  r_buffer[256]
;     //
;     // Write State variables
;     //
;     u1  w_state[8]
;     //
;     // Keep track of how many times we have been opened
;     //
;     u1  reference_count
;     //
;     // The driver number we loaded into
;     //
;     u1  driver_num
; }


PIPE_FLAGS       equ 0
PIPE_SIZE        equ PIPE_FLAGS+1
PIPE_START       equ PIPE_SIZE+1
PIPE_BUFFER      equ PIPE_START+1
PIPE_R_LEN       equ PIPE_BUFFER+256
PIPE_R_RESV      equ PIPE_R_LEN+1
PIPE_R_BUFFER    equ PIPE_R_LEN+3
PIPE_W_STATE     equ PIPE_R_BUFFER+256
PIPE_REFCOUNT    equ PIPE_W_STATE+8
PIPE_DRVNUM      equ PIPE_REFCOUNT+1
PIPE_END         equ PIPE_DRVNUM+1


;
; Used for denoting a write in progress
; in the flags byte
;
PIPE_WRITE_BIT  equ     acc.7


;
; Get the pipe buffer from TINIOS
```

```
;
PIPE_GET_BUFFER MACRO
        mov   dptr, #QueueID           ; see if we have an ESB installed
        lcall NatLib_GetEphemeralStateBlock
ENDM


;
; Debug Macro
;
PIPE_DEBUG MACRO PARAM value
        PUSH_DPTR1
        push acc
        push    dps
        mov     dps,#0
        mov     dptr,#Debug_Str
        lcall   info_sendstring
        mov     a,#value
        lcall   info_sendtwohex
        lcall   info_sendcrlf
        pop     dps
        pop     acc
        POP_DPTR1
ENDM
```

Appendix C: PipeDriver.java

```
package com.dalsemi.pipeio;

import com.dalsemi.comm.*;

public class PipeDriver
{
        static int pipe_port;
        //
        // This method dynamically looks up the
        // next available driver from the system
        // and allocates it, or uses the current
        // driver number if the pipe is
        // already loaded.
        //
        static native int getPipeDriver();

        static
        {
                System.loadLibrary("driver.tlib");
                pipe_port = getPipeDriver();
        }
}
```

Appendix D: PipeInputStream.java

```
package com.dalsemi.pipeio;

import com.dalsemi.comm.*;
import java.io.*;

public class PipeInputStream extends InputStream
{

        //
        // Handle to the pipe input stream
        //
        int pipe_handle;
        //
        // Input buffer for single byte reads
        //
        byte [] charArr;


        public PipeInputStream() throws Exception
        {
                //
                // Denote the pipe as invalid
                //
                pipe_handle = -1;
                charArr = new byte[1];
                try
                {
                        //
                        // On any exception, make the pipe invalid
                        //
                        pipe_handle = NativeComm.open(PipeDriver.pipe_port,
                                NativeComm.STREAM_STDIN);
                }
                catch (Exception E)
                {
                        pipe_handle = -1;
                        throw E;
                }
        }

        public int read(byte arr[], int offset, int length) throws IOException
        {
                //
                // Perform a validity check
                //
                if (pipe_handle == -1)
                        throw new IOException("pipe is not valid");
                //
                // Perform a length/offset check. This is *not* done
                // by I/O for you.
                //
                if ((offset >= 0) && ((offset+length) <= arr.length))
                {
                        return NativeComm.read(
                                pipe_handle,arr,offset,length,0,false);
                }
                else throw new ArrayIndexOutOfBoundsException(
                        "offset = " + offset + "length = " + length);
        }
```

```
        public int read() throws IOException
        {
                int len;
                //
                // Perform a validity check
                //
                if (pipe_handle == -1)
                        throw new IOException("pipe is not valid");
                //
                // Read a single byte
                //
                len = NativeComm.read(pipe_handle,charArr,0,1,0,false);

                return ((len==1)?((int)charArr[0]&0xFF):-1);
        }

        public void close() throws IOException
        {
                if (pipe_handle != -1)
                        NativeComm.close(pipe_handle);
        }
}
```

Appendix E: PipeOutputStream.java

```
package com.dalsemi.pipeio;

import com.dalsemi.comm.*;
import java.io.*;

public class PipeOutputStream extends OutputStream
{
        //
        // Handle to the pipe input stream
        //
        int pipe_handle;
        //
        // Array for passing single byte values
        //
        byte [] charArr;

        public PipeOutputStream() throws Exception
        {
                //
                // Denote the pipe as invalid
                //
                pipe_handle = -1;
                charArr = new byte[1];
                try
                {
                        //
                        // On any exception, make the pipe invalid
                        //
                        pipe_handle = NativeComm.open(PipeDriver.pipe_port,
                                NativeComm.STREAM_STDOUT);
                }
                catch (Exception E)
                {
                        pipe_handle = -1;
                        throw E;
                }
        }

        public void write(byte [] arr, int offset, int len) throws IOException
        {
                //
                // Perform a validity check
                //
                if (pipe_handle == -1)
                        throw new IOException("pipe is not valid");
                //
                // Perform a length/offset check. This is *not* done
                // by I/O for you.
                //
                if ((offset >= 0) && ((offset+len) <= arr.length))
                {
                        NativeComm.write(pipe_handle,arr,offset,len);
                }
                else
                        throw new ArrayIndexOutOfBoundsException(
                                "offset = " + offset + " len = " + len);
        }

        public void write(int x) throws IOException
```

```
        {
                //
                // Perform a validity check
                //
                if (pipe_handle == -1)
                        throw new IOException("pipe is not valid");

                charArr[0] = (byte)(x & 0xFF);

                NativeComm.write(pipe_handle,charArr,0,1);
        }

        public void close() throws IOException
        {
                if (pipe_handle != -1)
                        NativeComm.close(pipe_handle);
        }

}
```