Chapter 5

Serial Drivers

- What’s different about serial drivers?
- Supporting both I/O System and target agent interfaces.
- Writing interrupt, polled and bi-modal drivers.
- Serial driver initialization.
- Serial reads and write.
- Serial driver ioctl’s.
5.1 Overview

ttyDrv

Writing xxDrv

Modifying sysSerial.c

Supporting WDB Communication

• What is different about a serial driver
• Supporting I/O System and target agent interfaces
• What is ttyDrv?
• Typical serial driver data flow.
What’s Different about Serial Drivers?

- Serial drivers may need to perform work not directly related to managing the device. E.g.:
  - Mapping <CR> to <CR><LF> on output
  - Provide command line editing capabilities
  - Buffering input and output data
- To use a serial driver as a communication mechanism between the Tornado target server (on the host) and target agent (on the target) requires a special interface.
Driver Options

- A serial driver is not required to use or do anything described in this chapter.
- Following the method described in this chapter may substantially decrease development time.
The role of \texttt{ttyDrv}

\texttt{ttyDrv} is a virtual driver that
\begin{itemize}
  \item Manages I/O system
  \item Calls real driver to manage hardware
\end{itemize}

\texttt{ttyDrv}:
\begin{itemize}
  \item Supports the standard I/O system interface
  \item Provides device independent serial driver functionality
  \item May drive multiple lower level serial drivers
  \item The virtual \texttt{ttyDrv} driver and the lower, real driver replaces the older (VxWorks 5.2 and earlier) \texttt{tyCoDrv}.
\end{itemize}
xxDrv supports:

- Standard I/O interface
- Target agent interface (communication with target server)
- Bi-modal operation (interrupt or poll mode)

Use of serial channel by the target agent requires reconfiguring: modify `wind/target/config/target/config.h`, remake the system image and reboot. See section 8.4.1 of the *VxWorks Programmer’s Guide* for details.

Shown above are the ways Tornado uses the serial driver. Alternate mechanisms could be provided by the user by providing their own callback mechanism to handle the passing of data from the device to higher level protocols.
Callbacks

- Allows higher level protocol to specify how the driver is to pass data to and from it.
- There are two callbacks:
  - put callback that the driver calls to pass a character read from the device to the higher level protocol
  - get callback to fetch a character from the higher level protocol in order to write it out to the device
How Is This Driver Different?

- Different entry points.
- Routines not installed in Driver Table.
- Does not add device descriptor in Device List.
- Library supplied to handle device-independent serial driver functionality, including:
  - `selectLib` processing
  - data buffering
  - standard ioctl commands (e.g. `FIOSETOPTION`)
  - I/O system interface (i.e. `xxOpen`, `xxRead`, etc.)
**The WRS Serial Model**

### Generic code

- **usrConfig.c**
- **ttyDrv**

### Implementation specific code

- **sysSerial.c**

### Device specific code

- **xxDrv.c**

**xxDrv** manages a specific serial chip. Information about the specific implementation is provided in **sysSerial.c**.

- **Generic code:**
  - system startup code in **wind/target/config/all/usrConfig.c**
  - device independent interface to serial I/O in **ttyDrv**

- Implementation specific code in **wind/target/config/target/sysSerial.c** to initialize the specific instance of driver (e.g. address of registers).

- **xxDrv** manages device.
Serial Drivers

Overview

5.2 ttyDrv

Writing xxDrv

Modifying sysSerial.c

Supporting WDB Communication

- ttyDrv’s interface to VxWorks I/O system
- ttyDrv initialization
ttyDrv is a virtual driver that manages the interaction between the I/O system and the lower level actual drivers (with support from tyLib).

- The initialization of ttyDrv is done during VxWorks startup in wind/target/config/all/usrConfig.c.
- tyLib is a support library that existed in earlier versions of VxWorks.
- tyLib manages:
  - input and output ring buffers
  - many standard serial ioctl() calls
  - selectLib
ttyDrv and the I/O System

ttyDrv (and tyLib) manage application level and I/O system communication, including:

- I/O System requirements (such as adding entry in Driver Table and creating device descriptor and adding it to the system Device List)
- Handling all I/O system entry routines, e.g. ttyOpen, ttyIoctl, tyRead, tyWrite.
- Manages selectLib calls
- Manages command line editing (see ioctl functionality in tyLib man page)
- Manages data buffering
- Manages task synchronization on full or empty buffers
- Manages mutual exclusion on buffers

• All contact with the I/O system is done through ttyDrv/tyLib:
  - ttyDrv handles open and ioctl (ttyOpen and ttyIoctl)
  - tyLib handles read and write (tyRead and tyWrite)
  - there is no driver close routine
The real serial driver (xxDrv) moves data between the I/O System and the device through callbacks installed by ttyDrv:

- **tyLib** provides the callback routines (tyITx and tyIRd)
- **ttyDevCreate** installs the callback routines
Overview of ttyDrv( )

STATUS ttyDrv (void)

- Calls iosDrvInstall( ) to installs ttyDrv in the driver table using both ttyDrv and tyLib entry points.
- Automatically called if INCLUDE_TYCODRV_5_2 is not defined.

- The ttyDrv( ) routine is conditionally called in wind/target/config/all/usrConfig.c.
ttyDevCreate( )

STATUS ttyDevCreate (devName, pSioChan, rdBufSize, wrtBufSize)

• Allocates and initializes a device descriptor
• Initializes tyLib by calling tyDevInit():
  • selectLib initialization
  • Creates input and output ring buffers
  • Creates semaphores
• Calls iosDevAdd() to add the device to the device list
• Installs tyLib routines as input and output callbacks
• Starts the device in interrupt mode.

• We will discuss the SIO_CHAN struct later.
• The ttyDevCreate( ) routine is called in wind/target/config/all/usrConfig.c.
Serial Driver Initialization Overview

- If using Tornado serial drivers (INCLUDE_TYCODRV_5_2 is not defined), initialization is done at system startup by `usrRoot` task:
  - `ttyDrv()` will be called
  - `ttyDevCreate()` will be called NUM_TTY times

- Steps to automate initialization:
  - Modify `NUM_TTY` in `wind/target/config/target/config.h` to reflect total serial channels
  - Modify the three routines in `wind/target/config/target/sysSerial.c`
/*ụsụRoot - the root task */

void ụsụRoot (...)
{
    ...
    #ifdef INCLUDE_IO_SYSTEM
    iosInit (NUM_DRIVERS, NUM_FILES, "\null");
    ...
    consoleFd = NONE; /* assume no console device */
    ...
    #ifdef INCLUDE_TYCODRV_5_2
    ...
    /* vxWorks 5.2 style serial driver code */
    ...
    #else /* !INCLUDE_TYCODRV_5_2 */
usrConfig.c (cont’d)

```c
15 #ifdef INCLUDE_TTY_DEV
16     if (NUM_TTY > 0)
17         {
18            ttyDrv(); /* install console driver */
19
20            /* create serial devices */
21            for (ix = 0; ix < NUM_TTY; ix++)
22                {
23                #if (define(INCLUDE_WDB) && (WDB_COMM_TYPE ==
24                        WDB_COMM_UDPL_SLIP))
25                   /* don’t use WDB’s channel */
26                   if (ix == WDB_TTY_CHANNEL)
27                       continue;
28                #endif
29                sprintf (tyName, “%s%d”, “/tyCo/”, ix);
30                (void) ttyDevCreate (tyName,
31                                sysSerialChanGet(ix), 512, 512);
```
usrConfig.c (cont’d)

33 if (ix == CONSOLE_TTY) /* init the tty
    console*/
34 {
35    strcpy (consoleName, tyName); consoleFd = 
36       open (consoleName, O_RDWR, 0);
37       (void) ioctl (consoleFd, FIOBAUDRATE,
38       CONSOLE_BAUD_RATE);
39       (void) ioctl (consoleFd, FIOSETOPTIONS,
40       OPT_TERMINAL);
41   }
42 }
43 #endif/* INCLUDE_TTY_DEV */
44 ...
45 #endif/* !INCLUDE_TYCODRV_5_2 */
46 ...
47 endif/* INCLUDE_IO_SYSTEM */
Serial Driver Writes

\[ \text{write}\() \rightarrow \text{tyWrite}\() \rightarrow \text{xxDrv} \]

- \textit{ttyDrv} installs \textit{tyWrite} in the Driver Table as the write routine.
- \textit{xxDrv} drains the output ring buffer to the serial channel.

- The \textit{tyWrite}() routine is part of \textit{tyLib}.

\textit{tyWrite}():
- Blocks if ring buffer is full
- Copies data from user buffer to ring buffer
- If not undergoing transmit cycle, calls \textit{xxDrv} routine to initiate a transmit cycle

- A transmit cycle consists of outputting a byte to the device and waiting for a transmit interrupt (usually a separate interrupt) to transmit the next byte. The transmit interrupt indicates that the device is ready to transmit another byte.

- The \textit{tyLib} routine \textit{tyITx}() is used to get characters from the output ring buffer
Reads

\[
\text{read() } \leftarrow \quad \text{tyRead()} \quad \rightarrow \quad \text{xxDrv}
\]

- \textit{ttyDrv( )} installs \textit{tyRead( )} into the Driver Table as the read routine.
- \textit{xxDrv} reads characters from the serial channel and inserts them into the input ring buffer.

- The \textit{tyRead( )} routine is part of \textbf{tyLib}.

\textit{tyRead( )}:
- Blocks if ring buffer is empty
- Copies bytes from ring buffer into user buffer
- Handles Xon/Xoff processing
- If there are still characters in the input ring buffer, unblocks any task sleeping on a read of the serial channel

- The \textbf{tyLib} routine \textit{tyIRd( )} is used to add characters into the input ring buffer.
Serial ioctl

ioctl()  \(\rightarrow\) ttyIoctl()  \(\rightarrow\) xxIoctl()  \(\rightarrow\) tyIoctl()

- Ioctl command is passed to driver xxIoctl().
- If driver fails request, ttyIoctl() calls tyIoctl().

- Ioctl commands supported by ttyIoctl():
  
  - FIONREAD: Returns the number of characters in the input (read) buffer
  - FIONWRITE: Returns the number of characters in the output (write) buffer
  - FIOFLUSH: Discards all characters in both buffers
  - FLOWFLUSH: Discards all characters in output buffer
  - FIORFLUSH: Discards all characters in the input buffer
  - FIOSETOPTIONS: Set device options
  - FIOGETOPTIONS: Returns current device options
  - FIOCANCELS: Cancels read or write
  - FIOISATTY: Returns TRUE
  - FIOSELECT: Add to the select list
  - FIOUNSELECT: Remove from the select list
ttyDrv and tyLib

usrRoot

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

ttyDrv

ttyDrv()
ttyDevCreate()
ttyOpen()
ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()

Application
open() ioctl() read() write()

I/O System
ttyOpen() ttyIoctl()
Serial Drivers

Overview

ttyDrv

5.3 Writing xxDrv

Modifying sysSerial.c

Modifying sysSerial.c

- Serial driver device descriptor
- Serial driver entry points
Coding Steps

- Initialization
  - Define the number of supported serial channels
  - Initialize your driver’s device descriptors
  - Write your device initialization code
- Write your entry point routines
- Write the ISRs to manage device
- Use `wind/target/src/drv/sio/templateSio.c` as a template.

- There are no `xxDrv` and `xxDevCreate` routines as this functionality is handled by `ttyDrv`. 
Serial drivers are initialized during the VxWorks system startup code.
The Device Descriptor

- \texttt{XX_DRV} struct has an \texttt{XX_CHAN} for each channel.
- Each \texttt{XX_CHAN} points to \texttt{SIO_DRV_FUNCS}.
- \texttt{SIO_DRV_FUNCS} has the driver's entry points.

- \texttt{XX_DRV} struct is the central data structure used by \texttt{xxDrv}.
- \texttt{XX_CHAN} struct contains:
  - channel specific information needed by \texttt{xxDrv}
  - pointer to the driver's \texttt{SIO_DRV_FUNCS} struct
- Example:

  ```c
  typedef struct
  {
    SIO_DRV_FUNCS * pFooDrvFuncs;
    ...
  } FOO_CHAN;

  typedef struct
  {
    FOO_CHAN portA;
    FOO_CHAN portB;
    ...
  } FOO_DRV;
  ```
XX_CHAN Example

typedef struct foo_chan
{
    /* must be first */
    SIO_DRV_FUNCS * pFooDrvFuncs;
    /* callbacks to higher level protocols */
    STATUS       (*getTxChar) ();
    STATUS       (*putRcvChar) ();
    void *       getTxArg;
    void *       putRcvArg;
    /* Device specific data */
    volatile char * cr; /* control register */
    ...
/* misc. */
    int mode;    /* interrupt or poll mode */
    int baudRate;
    BOOL isrInstalled;
    ...
} FOO_CHAN;
The SIO_CHAN Struct

- For a regular character driver, a pointer to a DEV_HDR is used to pass info between driver and I/O system.
- For a serial driver, a pointer to a SIO_CHAN is used to pass info between driver and higher level protocols:

```c
typedef struct sio_chan {
    SIO_DRV_FUNCS * pDrvFuncs; /* device data */
} SIO_CHAN;
```

- Defined in wind/target/h/sioLib.h:

```c
typedef struct sio_chan {
    SIO_DRV_FUNCS * pDrvFuncs;
    /* device data */
} SIO_CHAN;
```

- Since the higher level protocols do not know the definition of the driver’s XX_CHAN struct, the SIO_CHAN struct is used to allow a well-defined data type to be exchanged between layers.
Communication between xxDrv and ttyDrv

- **ttyDrv** requests services from **xxDrv** through the entry points provided in `SIO_DRV_FUNCS`.
- **xxDrv** requests services from **ttyDrv** through callbacks
  - a callback is a pointer to a function
  - using callbacks allows **xxDrv** to dynamic switch between communicating with **ttyDrv** and the target agent
## Entry Points

The SIO_DRV_FUNCS struct contains the addresses of the driver entry points

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxCallBackInstall()</td>
<td>Installs access to higher level protocols (i.e., I/O system or target agent)</td>
</tr>
<tr>
<td>xxPollOutput()</td>
<td>Poll mode output routine</td>
</tr>
<tr>
<td>xxPollInput()</td>
<td>Poll mode input routine</td>
</tr>
<tr>
<td>xxIoctl()</td>
<td>Support device specific ioctl cmds</td>
</tr>
<tr>
<td>xxTxStartup()</td>
<td>Initiates a transmit cycle</td>
</tr>
</tbody>
</table>
Driver Callback Install Routine

```c
int xxCallbackInstall (pSioChan, callbackType, callback, callbackArg)
```

- `pSioChan` pointer to `SIO_CHAN` struct
- `callbackType` `SIO_CALLBACK_GET_TX_CHAR` or `SIO_CALLBACK_PUT_RCV_CHAR`
- `callback` pointer to callback routine
- `callbackArg` argument to pass callback

- Initialize the appropriate members in the `SIO_CHAN` struct
- Return `OK` or `ENOSYS` if callback type is not one of the two above
Example Driver Install Callback Routine

```c
1 static int fooCallbackInstall
2   ( 
3   SIO_CHAN * pSioChan,
4   int    callbackType,
5   STATUS (*callback)(),
6   void *  callbackArg
7 )
8
9 { 
10  FOO_CHAN * pFooChan = (FOO_CHAN *) pSioChan;
11  switch (callbackType) 
12   { 
13       case SIO_CALLBACK_GET_TX_CHAR: 
14           pFooChan->getTxChar = callback;
15           pFooChan->getTxArg = callbackArg
16           return (OK);
17   }
```
Example Driver Install Callback (con'td)

18     case SIO_CALLBACK_PUT_RCV_CHAR:
19         pFooChan->putRcvChar = callback;
20         pFooChan->putRcvArg = callbackArg
21             return (OK);
22     default:
23         return (ENOSYS);
24     }
25 }


Driver Initialization

void xxDevInit (pXxDrv)

- Argument is a pointer to the xx_DRV struct.
- Initialize XX_CHAN:
  - the SIO_DRV_FUNC struct with your routines
  - the callback routines to dummy routines
  - any driver specifics
- Reset chip.
Example xxDevInit

```c
1 static SIO_DRV_FUNCS fooSioDrvFuncs =
2 {
3     fooIoctl,              
4     fooTxStartup,        
5     fooCallbackInstall, 
6     fooPollInput,        
7     fooPollOutput        
8     }
9
10 void fooDevInit (FOO_DRV * pFooDrv) 
11 {                          
12     /* initialize driver function pointers */
13     pFooDrv->portA.pDrvFuncs      = &fooSioDrvFuncs;
14
15     /* install dummy driver callbacks */
16     pFooDrv->portA.getTxChar     = fooDummy;
17     pFooDrv->portA.putRcvChar    = fooDummy; 
18```
Example xxDevInit (cont’d)

19 /* reset the chip */
20     *pFooDrv->masterCr = FOO_RESET_CHIP;
21
22 /* setting polled mode is one way to make
   the device quiet */
23     fooIoctl ((SIO_CHAN *)&pFooDrv->portA,
24         SIO_MODE_SET, (void *) SIO_MODE_POLL);
25 
26
27 static int fooDummy (void)
28 {
29     return (ERROR);
30 }
Writes

- User `write( )` goes through I/O system and calls `tyWrite( )` — ttyDrv’s write entry point in the driver table.
- The `tyWrite( )` copies data into a ring buffer and calls driver’s `xxTxStartup( )` to initiate a transmit cycle.
- On transmit interrupts, driver’s `xxTxInt( )` called.

- The `tyWrite` routine is part of `tyLib` (not actually `ttyDrv`).
Driver xxTxStartup( ) Routine

void xxTxStartup (pSioChan)

- Initiate a transmit cycle by writing a byte fetched from higher level protocol to your device.
- Turn on transmit interrupts, if needed.
- Byte from higher level protocol is fetched by calling the getTxChar callback initialized by ttyDrv.
Example Startup Routine

```
static int fooTxStartup (SIO_CHAN * pSioChan)
{
    FOO_CHAN * pFooChan = (FOO_CHAN *)pSioChan;
    char outChar;

    if ((*pFooChan->getTxChar)
        (pFooChan->getTxArg, &outchar) != ERROR)
    {
        *pFooChan->dr = outChar;
        /* Turn on transmitter interrupts, if needed */
        *pFooChan->cr |= TX_INT_ENABLE;
    }
    return (OK);
}
```
The Transmit ISR

void xxIntTx (pFooChan)

- Fetch byte from higher protocol by calling the `getTxChar` callback.
- Write byte out to device.
- Clear interrupt (if needed).
- Reset transmit interrupts if no byte is available (if needed).
Transmit ISR Example

1    void fooIntTx (FOO_CHAN * pFooChan)
2    {
3        char outChar;
4
5        if ((*pFooChan->getTxChar)
6            (*pFooChan->getTxArg, &outChar) != ERROR)
7            *pFooChan->dr = outChar;
8        else
9            /* Reset interrupt, if needed */
10           *pFooChan->cr = FOO_RESET_TX;
11
12       /* acknowledge interrupt if needed */
13       *pFooChan->cr = FOO_RESET_INT;
14    }
void xxIntRcv (pSioChan)

- Read byte from device.
- Pass byte to higher level protocol by calling *putRcvChar* callback.
Receive ISR Example

```c
1  void fooIntRcv (FOO_CHAN * pFooChan)
2  {
3      char inChar;
4
5      inChar = *pFooChan->dr;
6     /* acknowledge interrupt if needed */
7      *pFooChan->cr = FOO_RESET_INT
8     (*pFooChan->putRcvChar)
9         (pFooChan->putRcvArg, inchar);
10  }
```
I/O Control

- User `ioctl()` calls (through the I/O system) `ttyioctl()` which calls driver’s ioctl.
- If driver does not handle command, it is passed on to `tyLib`

- See `tyLib` man page for ioctl handled by `tyLib`.
- The `ttyioctl()` routine:
  - re-maps FIOBAUDRATE to SIO_BAUD_RATE
  - calls driver `xxioctl()`
  - if driver’s ioctl returns ENOSYS, calls `tyioctl()`
**Driver ioctl()**

- Standard ioctl commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIO_BAUD_SET</td>
<td>Set baud rate</td>
</tr>
<tr>
<td>SIO_BAUD_GET</td>
<td>Return baud rate</td>
</tr>
<tr>
<td>SIO_MODE_SET</td>
<td>Set interrupt/poll mode</td>
</tr>
<tr>
<td>SIO_MODE_GET</td>
<td>Return current mode</td>
</tr>
<tr>
<td>SIO_AVAIL_MODES_GET</td>
<td>Return available modes</td>
</tr>
<tr>
<td>SIO_HW_OPTS_SET</td>
<td>Set hardware options</td>
</tr>
<tr>
<td>SIO_HW_OPTS_GET</td>
<td>Get hardware options</td>
</tr>
</tbody>
</table>

- If command is not recognized, return **ENOSYS**.
Example ioctl

1 static int fooIoctl
2   (SIO_CHAN * pChan, int command, void * arg)
3 {
4   FOO_CHAN * pFooChan = (SIO_CHAN *)pChan;
5
6   switch (command) {
7     case SIO_BAUD_SET:
8       if (arg < FOO_MIN_BAUDRATE ||
9           arg > FOO_MAX_BAUDRATE)
10          return (EIO);
11     ...
12       pFooChan->baudrate = baudrate;
13     return (OK);
Example ioctl (cont'd)

19 case SIO_BAUD_GET:
20    *(int *)arg = pFooChan->baudrate;
21    return (OK);
22 case SIO_MODE_SET:
23    return (fooModeSet (pFooChan,
24             (uint_t)arg) == OK ? OK : EIO);
25 case SIO_MODE_GET:
26    *(int *)arg = pFooChan->mode;
27    return (OK);
28 case SIO_AVAIL_MODES_GET:
29    *(int *)arg = SIO_MODE_INT |
30        SIO_MODE_POLL;
31    return (OK);
32 case SIO_HW_OPTS_SET:
33 case SIO_HW_OPTS_GET:
34    return (ENOSYS);
35    default: return (ENOSYS);
36    }
37    }

ioctl Support Routine

38 static int fooModeSet
39 (  
40   FOO_CHAN * pFooChan,
41   uint_t    newMode
42 )
43 {
44   switch (newMode)
45   {
46     case SIO_MODE_INT:
47       /* fail request for interrupt mode if ISRs
48          are not yet installed */
49       if (pFooChan->isrInstalled == FALSE)
50         return (EIO);
51       *pFooChan->cr |= FOO_INT_ENABLE;
52     break;
ioctl Support Routine (cont'd)

```c
52  case SIO_MODE_POLL:
53       *pFooChan->cr &= ~FOO_INT_ENABLE;
54       break;
55     default:
56        return (EIO);
57     }
58
59     pFooChan->mode = newMode;
60     return (OK);
61   }
```
The Hdw Options ioctl Cmd

- The SIO_HW_OPTS_SET and SIO_HW_OPTS_GET are designed to allow a standard (POSIX) way or requesting hardware options (parity, stop bits, etc.)
- Support is optional; return ENOSYS if not supported.
- Options are bitwise ORed together.
- It is the caller’s responsibility to make a get call after a set to see which options are implemented, i.e. implement those options you can or want to implement.

Example user interface:

```c
int options;
...
if (ioctl (fd, SIO_HW_OPTS_SET, PARENB | PARODD) == OK)
{
    /* Setting hardware options supported */
ioctl (fd, SIO_HW_OPTS_GET, &options);
    if (options & PARENB)
        { /* parity supported */ }
    if (options & PARODD)
        { /* odd parity supported */ }
}
else
{
    /* driver does not support setting hardware options */
}
```
## Hardware Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCAL</td>
<td>ignore modem status lines</td>
</tr>
<tr>
<td>CREAD</td>
<td>enable device receiver</td>
</tr>
<tr>
<td>CS[5-8]</td>
<td>set char size to 5, 6, 7 or 8 bits</td>
</tr>
<tr>
<td>HUPCL</td>
<td>hang up on last close</td>
</tr>
<tr>
<td>STOPB</td>
<td>2 stop bits (else one)</td>
</tr>
<tr>
<td>PARENB</td>
<td>enable parity (else disabled)</td>
</tr>
<tr>
<td>PARODD</td>
<td>odd parity (else even)</td>
</tr>
</tbody>
</table>

- Default options:
  - eight data bits
  - no parity
  - one stop bit

- That is:

\[(\text{CLOCAL} \mid \text{CREAD} \mid \text{CS8}) \& \sim(\text{HUPCL} \mid \text{STOPB} \mid \text{PARENB})\]
Serial Drivers

Overview

ttyDrv

Writing xxDrv

5.4 Modifying sysSerial.c

Supporting WDB Communication

- Serial driver poll routines
- Providing bi-modal support
sysSerial.c

- **sysSerialHwInit()**
  - Performs device initialization
  - Called by **sysHwInit()** in **usrInit()**
- **sysSerialHwInit2()**
  - Installs ISRs
  - Called by **sysClkConnect()** in **usrRoot** task
- **sysSerialChanGet()**
  - Converts integer channel number to pointer to device descriptor
  - Used by **usrRoot()** to obtain channel pointer to pass to **ttyDevCreate()**
  - Used by target agent to obtain channel pointer
Initialization

- Modify \texttt{NUM_TTY} in \texttt{wind/target/config/target/config.h}.
- Declare device descriptor structures in your own \texttt{xxDrv.h}.
- Modify \texttt{wind/target/config/target/sysSerial.c}
  - Initialize device descriptors
  - reset serial device
  - install ISRs
  - convert channel number to a pointer to device descriptor
sysSerialChanGet( )

SIO_CHAN * sysSerialChanGet (chan)

• Converts an integer channel number to a pointer to a SIO_CHAN structure.

• Example:

```c
SIO_CHAN * sysSerialChanGet (int chan)
{
    switch (chan)
    case 0:
        return ((SIO_CHAN *)&fooDrv.portA);
    case 1:
        return ((SIO_CHAN *)&fooDrv.portB);
    default:
        return ((SIO_CHAN *) ERROR);
}
```

• Located in sysSerial.c in the target directory.
void sysSerialHwInit (void)

- Initialize the hardware dependent portion of the driver’s `XX_CHAN` struct (e.g. register addresses).
- Call `xxDevInit()` to perform the actual device initialization.
- Is called by `sysHwInit()` during system startup from the `usrInit()` (before kernel initialization).
- Set flag to indicate ISR are not yet installed.

- Located in `sysSerial.c` in the target directory.

- Example:

  ```c
  void sysSerialHwInit (void)
  {
    fooDrv.portA.cr = FOO_PORTA_CR;
    fooDrv.portB.cr = FOO_PORTB_CR;
    ...
    fooDrv.portA.isrInstalled = FALSE;
    fooDrv.portB.isrInstalled = FALSE;
    fooDevInit (&fooDrv);
  }
  ```
void sysSerialHwInit2 (void)

- Connect interrupt service routines
- The 3rd arg for `intConnect()` should be a pointer to your `XX_CHAN` struct: e.g.
  ```
  intConnect (vector, fooIsr, (int)pFooChan);
  ```
- Is called by `sysClkConnect()` during system startup from usrRoot task (after system has been sufficiently initialized to install ISRs)
- Set flag to indicate ISRs are installed.

- Located in `sysSerial.c` in the target directory.
- Example:
  ```
  void sysSerialHwInit2 (void)
  {
      (void) intConnect (INUM_TO_IVEC (INT_VEC_FOO_PORTA_RD),
                        fooISRRd, (int) &fooDrv.portA);
      ...
      fooDrv.portA.isrInstalled = TRUE;
  }
  ```
Serial Drivers

Overview

ttyDrv

Writing xxDrv

Modifying sysSerial.c

5.5 Supporting WDB Communication

- Serial driver poll routines
- Providing bi-modal support
Overview

- Three communication modes:
  - interrupt mode which allows task level debugging
  - poll mode which allows system level debugging
  - bi-modal which allows switching between interrupt and poll mode
- To support interrupt mode, no further development necessary
- To support poll mode, must write input and output poll routines
- Support is optional (but recommended).
Not Supporting Poll Mode

If not supporting poll mode:

- Write a dummy poll routine which just returns ERROR.
- Initialize the xxPollInput and xxPollOutput members of your SIO_DRV_FUNCS struct to the dummy routine.
- Modify xxIoctl() to reflect support:
  - The SIO_MODE_GET should just set arg to SIO_MODE_INT
  - The SIO_MODE_SET should return EIO if arg is SIO_MODE_POLL
Supporting Poll Mode

To support poll mode:

• write `xxPollInput()`

• write `xxPollOutput()`

• Modify `xxIoctl()` commands to reflect poll support:
  • SIO_MODE_GET
  • SIO_MODE_SET
  • SIO_AVAIL_MODES_GET
Poll Mode: Input

int xxPollInput (pSioChan, pChar)

• If input character available, read it from device and put the character at the address of \texttt{pChar} and return \texttt{OK}

• If no character is available, return \texttt{EAGAIN}
Example Input Poll Routine

```c
1 static int fooPollInput
2  ( 
3    SIO_CHAN *   pSioChan,
4    char *       pInputChar
5  )
6 {
7    FOO_CHAN *pFooChan = (FOO_CHAN *) pSioChan;
8    if ((pFooChan->sr & FOO_RX_AVAIL) == 0)
9      /* no char available */
10     return (EAGAIN);
11
12    /* char available */
13    *pInputChar = pFooChan->dr;
14    return (OK);
15 }
```
Output Poll Routine

```c
int xxPollOutput (pSioChan, outChar)
```

- If device ready/able to receive char, write char to device and return OK
- If device can not receive char, return EAGAIN
Example Poll Output Routine

```c
1 static int fooPollOutput
2   (SIO_CHAN * pSioChan,
3      char outchar)
4 {
5     FOO_CHAN * pFooChan = (FOO_CHAN *)pSioChan;
6     if ((*pFooChan->sr & FOO_TX_READY) == 0)
7         return (EAGAIN); /* can’t send */
8     /* OK to send char */
9     *pFooChan->dr = outChar;
10    return (OK);
11 }
```
Implementing A Bi-Modal Driver

- Writing a driver which supports dynamically switchable modes allows use by both the external debug agent for system-level debugging and task-level debugging.

- To support dynamic switching:
  - Implement interrupt driver
  - Implement poll driver
  - Modify \texttt{xIoctl()} to reflect support of both:
    \begin{verbatim}
    SIO_AVAIL_MODES_GET
    SIO_MODE_GET
    SIO_MODE_SET
    \end{verbatim}
Caveat

- To support dynamical mode switches, your driver must be able to handle a mode switch while in the receive callback to the higher level protocols.
- For example, while in the receive callback in interrupt mode, the target agent might:
  - lock interrupts
  - switch to poll mode
  - use the device in poll mode
  - unlock interrupts
  - switch back to interrupt mode
The Issue

- Your receive interrupt handler (*xxIntRcv*), must insure that the interrupt has been completely managed before calling the *putRcvChar* callback routine.

- Example:

  ```c
  void fooIntRcv (...) {
    inChar = *pFooChan->dr;
    *pFooChan->cr = FOO_RESET_INT;
    (*pFooChan->putRcvChar) (...
  }
  ```

  Note that the interrupt acknowledgment must be done **before** the call to the *putRcvChar* callback.
Summary

• Write sequence
  • `write( ) ⇒ tyWrite( ) ⇒ xxTxStartup( )`
  • device transmit interrupt ⇒ `xxIntTx( ) ⇒ getTxChar`
callback

• Read sequence
  • `read( ) ⇒ tyRead( )`
  • device receive interrupt ⇒ `xxIntRcv( ) ⇒ putRcvChar`
callback

• Ioctl sequence
  • `ioctl( ) ⇒ ttyIoctl( ) ⇒ xxIoctl( )`
    ⇒ (if `xxIoctl( ) returns ENOSYS) `tyIoctl( )`
Summary

• Begin with the template code in wind/target/src/drv/sio/templateSio.c.

• Modify NUM_TTYS in config.h.

• Modify routines in sysSerial.c:
  • `sysSerialHwInit()`
  • `sysSerialHwInit2()`
  • `sysSerialChanGet()`

• Write initialization code: `xxDevInit()`.

Tornado Device Driver Workshop © Copyright Wind River Systems 5-71
Wind River Systems
Summary

• Write driver entry points:
  • \texttt{xxCallBackInstall()} 
  • \texttt{xxPollOutput()} 
  • \texttt{xxPollInput()} 
  • \texttt{xxIoctl()} 
  • \texttt{xxTxStartup()}
## Initialization Sequence

usrInit()
- ➔ sysHwInit()
  - ➔ sysSerialHwInit()
    - ➔ xxDevInit()
  - ➔ kernelInit()

usrRoot() /* first task */
- ➔ sysClkConnect()
  - ➔ sysSerialHwInit2()
- ➔ ttyDrv()
- ➔ sysSerialChanGet()
- ➔ ttyDevCreate()
  - ➔ xxCallbackInstall()
  - ➔ xxIoctl()