DeviceMaster® Nserial Device Driver

This document describes the **nserial** device driver for eCos. The driver provides a high-level API that user applications can use to access the DeviceMaster serial ports.

The **nserial** driver is based on the eCos 1.3.1 serial driver. The following enhancements have been made to the 1.3.1 serial driver:

- Non-blocking read/write
- Semi-blocking read
- Modem control/status line support
- Error count query
- Buffer status query
- Arbitrary baud rate
- Flow control enhancements

If you are unfamiliar with the eCos I/O device API, see the eCos Reference Manual, Part IV: I/O Package (Device Drivers).

nserial API

The nserial API consists of the five standard eCos system calls:

- **cyg_io_lookup**()
- **cyg_io_read**()
- **cyg_io_write**()
- **cyg_io_get_config**()
- **cyg_io_set_config**()

If you are unfamiliar with these calls please see the "User API" section of the eCos Reference Manual, Part IV: I/O Package (Device Drivers).

cyg_io_lookup()

The device names supported by the nserial driver are "**/dev/ser0**" through "**/dev/ser31**" (assuming a 32-port device).

cyg_io_read()

Reads may be blocking, non-blocking, or semi-blocking.

blocking The default state for a serial port is blocking reads. If **N** bytes are requested in the call to **cyg_io_read()**, the call will not return until N bytes have been transfered to the user's buffer. The call will wait indefinitely for the requested number of bytes.

non-blocking The call to **cyg_io_read()** will return immediately. If less than the requested number of bytes is available, the returned length will reflect the number of bytes actually transferred, and the status will return -**EAGAIN**.

semi-blocking The call to **cyg_io_read()** will return as soon as some data has been transferred. If no data is available, it will block indefinitely until at least one byte of data is available. If less than the requested number of bytes is read, the returned length will reflect the number of bytes actually transferred, and the status will return -**EAGAIN**.

The low-level UART driver in the 4/8/16/32 port DeviceMaster transfers data every 10ms, so a loop containing a semi-blocking **cyg_io_read()** call will execute once every 10ms while data is being received, and block indefinitely if no data is being received.

Other UART drivers may transfer data based on a FIFO full trigger rather than on a fixed time period. For example a loop containing a semi-blocking **cyg_io_read()** call might execute once for every 64 receive bytes.

cyg_io_write()

Writes may be blocking or non-blocking. There is no semi-blocking mode for write operations.

Blocking The call to **cyg_io_write()** will return as soon as all requested data has been transferred to the transmit buffer.

If there is insufficient room in the buffer, the call will block indefinitely until room becomes available.

non-blocking The call to **cyg_io_write()** will return immediately. If there was insufficient room in the transmit buffer for the requested number of bytes, the length will reflect the actual number of bytes transferred, and the status will return -**EAGAIN**.

cyg_io_get_config()

The following keys are supported for the **cyg_io_get_config()** call:

CYG_IO_GET_CONFIG_SERIAL_INFO
Gets the current configuration of the serial port. The structure in which data is returned is:

```
typedef struct {
    cyg_serial_baud_rate_t   baud;
    cyg_serial_stop_bits_t   stop;
    cyg_serial_parity_t      parity;
    cyg_serial_word_length_t word_length;
    cyg_uint32               flags;
    cyg_uint8                rxflow_xon_char;
    cyg_uint8                rxflow_xoff_char;
    cyg_uint8                txflow_xon_char;
    cyg_uint8                txflow_xoff_char;
    cyg_uint8                interface_mode;
    cyg_uint32               read_timeout;
    cyg_uint32               write_timeout;
    cyg_uint32               inter_char_timeout;
    cyg_serial_modem_callback_t *modem_callback;
    void                        *modem_callback_data;
    cyg_serial_rx_callback_t    *rx_callback;
    void                        *rx_callback_data;
    cyg_uint8                eol1;
    cyg_uint8                eol2;
} cyg_serial_info_t;
```

The fields are defined as follows:

baud The current baud rate. This might be one of the enumerated constant values defined in serialio.h such as **CYGNUM_SERIAL_BAUD_9600** or it may be the actual baud rate (e.g. the integer value 9600).

If the numerical value is less than 50, it is assumed to be an enum constant. If it is 50 or larger, it is assumed to be the actual baud rate.

stop One of the enumerated constant values defined in **serialio.h**:

```
CYGNUM_SERIAL_STOP_1
CYGNUM_SERIAL_STOP_1_5
CYGNUM_SERIAL_STOP_2
```

parity One of the enumerated constant values defined in **serialio.h**:

```
CYGNUM_SERIAL_PARITY_NONE
CYGNUM_SERIAL_PARITY_EVEN
CYGNUM_SERIAL_PARITY_ODD
CYGNUM_SERIAL_PARITY_MARK
CYGNUM_SERIAL_PARITY_SPACE
```

Note: *Not all UART types support mark and space parity. The DeviceMaster 4/8/16/32-port hardware only supports none/even/odd.*

word_length One of the enumerated constant values defined in **serialio.h**:

```
CYGNUM_SERIAL_WORD_LENGTH_5
CYGNUM_SERIAL_WORD_LENGTH_6
CYGNUM_SERIAL_WORD_LENGTH_7
CYGNUM_SERIAL_WORD_LENGTH_8
```

Note: *Not all UART types support 5 and 6 bit word lengths. The DeviceMaster 4/8/16/32-port hardware only supports 7 and 8 bit word lengths.*

flagsBit map of binary option flags:

CYG_SERIAL_FLAGS_RTSCTS enables RTS/CTS hardware flow control. Equivalent to setting both of **RTS_RX_FLOW** and **CTS_TXFLOW**.

CYG_SERIAL_FLAGS_RTS_RXFLOW
Enables hardware RTS flow control for transmitter.

CYG_SERIAL_FLAGS_CTS_TXFLOW
Enables hardware CTS flow control for receiver.

CYG_SERIAL_FLAGS_XONXOFF_RXFLOW
Enables Xon/Xoff flow control of receive data stream.

CYG_SERIAL_FLAGS_XONXOFF_TXFLOW
Enables Xon/Xoff flow control of transmit data stream.

CYG_SERIAL_FLAGS_RTS_TOGGLE
Enables automatic control of RTS for half-duplex communications. RTS is asserted while data is being transmitted and de-asserted when no data is being transmitted. This mode should be enabled when using the port in RS-485 mode or when using an external half-duplex RS-232 device such as a Bell-202 modem or an RS-232/485 converter.

CYG_SERIAL_FLAGS_EN_LOOPBACK
Enables hardware loopback of transmit data to receive data.

CYG_SERIAL_FLAGS_IGN_NULLS
Causes null receive bytes to be discarded.

CYG_SERIAL_FLAGS_RD_NONBLOCK
enables non-blocking read mode.

CYG_SERIAL_FLAGS_RD_SEMIBLOCK
Enables semi-blocking read mode.

CYG_SERIAL_FLAGS_WR_NONBLOCK
Enables non-blocking write mode.

rxflow_xoff_char The character transmitted when the receive buffer is nearly full and the driver wishes the remote device to stop sending.

rxflow_xon_char The character transmitted when the receive buffer empties and an Xoff has previously been sent to stop the remote device from sending.

txflow_xoff_char Receipt of this character will cause the driver to stop sending data until the Xon character is received.

txflow_xon_char Receipt of this character will cause the driver to resume sending data.

interface mode

```
#define RS232_MODE 0x00
#define RS422_MODE 0x08
#define RS485_MODE 0x10
```

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_DRAIN
Blocks until all buffered write data has been sent.

CYG_IO_GET_CONFIG_SERIAL_INPUT_FLUSH
Discards all buffered but unread receive data.

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_FLUSH
Discards all buffered but unsent write data.

CYG_IO_GET_CONFIG_SERIAL_ABORT
TBD

CYG_IO_GET_CONFIG_SERIAL_TX_STOP
Stop sending data, but do not discard buffered write data.

CYG_IO_GET_CONFIG_SERIAL_TX_START
Start sending buffered write data.

CYG_IO_GET_CONFIG_SERIAL_BUFFER_INFO
Gets the current status of the read and write buffers. The structure in which data is returned is:

```
typedef struct {
   cyg_int32 rx_bufsize;
   cyg_int32 rx_count;
   cyg_int32 tx_bufsize;
   cyg_int32 tx_count;
 } cyg_serial_buf_info_t;
```

These values do not include any data buffered in the UART. Only data immediately accessible to **cyg_io_read()** and buffer space immediately usable by **cyg_io_write()** is reported.

CYG_IO_GET_CONFIG_SERIAL_MODEM
Gets the current state of the modem control and status lines. Data is returned as an unsigned int bitmap with masks defined in **serialio.h**:

```
CYG_SERIAL_MODEM_CTS
CYG_SERIAL_MODEM_DSR
CYG_SERIAL_MODEM_RI
CYG_SERIAL_MODEM_CD
CYG_SERIAL_MODEM_CTS_DELTA
CYG_SERIAL_MODEM_DSR_DELTA
CYG_SERIAL_MODEM_RI_DELTA
CYG_SERIAL_MODEM_CD_DELTA
CYG_SERIAL_MODEM_DTR
CYG_SERIAL_MODEM_RTS
CYG_SERIAL_MODEM_FLOWTXOFF
```

CYG_IO_GET_CONFIG_SERIAL_ERRORS
Gets the count of various types of serial errors. The structure in which data is returned is:

```
typedef struct {
  unsigned rxOverflow;
  unsigned rxParity;
  unsigned rxFraming;
  unsigned rxBreak;
  unsigned fifo;
} cyg_serial_error_count_t;
```

CYG_IO_GET_CONFIG_SERIAL_CLRERRS
Resets the error counts to zero.

CYG_IO_GET_CONFIG_SERIAL_BREAKON
Enables the "*send break*" feature of the UART. This will cause a space to be sent continuously until the **BREAKOFF** key is invoked.

CYG_IO_GET_CONFIG_SERIAL_BREAKOFF
Disables the "send break" feature of the UART. This returns the UART to normal data transmission mode.

CYG_IO_GET_CONFIG_TX_OVERRIDE
Clears any xoff condition that may have been set by receipt of an Xoff character while Xon/Xoff transmit flow control is enabled.

cyg_io_set_config()

The following keys are supported for the **cyg_io_set_config()** call:

- **CYG_IO_SET_CONFIG_SERIAL_INFO**
- **CYG_IO_SET_CONFIG_SERIAL_WRITE_PRIORITY_BYTE**
- **CYG_IO_SET_CONFIG_SERIAL_MODEM**

Example Program

The following example program uses the nserial driver API to echo received data on all 16 ports of a DeviceMaster 16-port platform.

```
#include <cyg/kernel/kapi.h>
#include <cyg/error/codes.h>
```

io.h and **serialio.h** declare the **nserial** driver API:

```
#include <cyg/io/io.h>
#include <cyg/io/serialio.h>
#include <stdio.h>
#include <stdlib.h>
```

The **nserial** driver **does _not_ know** about the selectible hardware interface feature of the DeviceMaster. That is handled below:

```
// macros to set hardware interface type

#define InterfaceSelect232  (0<<3)
#define InterfaceSelect422  (1<<3)
#define InterfaceSelect485  (2<<3)
#define InterfaceSelect232i (3<<3)
```

(Use 232i mode when you want to use the RTS toggle feature with an RS-232 interface.)

The following is used to insure that output from different calls to **diag_printf_m()** don't get interlaced in the diag UART output data stream. The **diag_printf()** function is a very low level, busy-wait, non-buffered, non-atomic output routine:

```
// Stuff to implement atomic diagnostic message output

// printf routine that prints messages to KS32C5000 UART
extern void diag_printf(const char *fmt, ...);

// atomic diag_printf operation -- only use in running tasks,
// not in initialization code, DSR or ISR code.

#define UseDiagPrintfMutex 1

#if UseDiagPrintfMutex
static cyg_mutex_t dpMutex;
#define diag_printf_m(fmt, args...) \
  do { \
       cyg_mutex_lock(&dpMutex); \
       diag_printf(fmt, ##args); \
       cyg_mutex_unlock(&dpMutex); \
     } while (0)
#define diag_init()  cyg_mutex_init(&dpMutex)
#define diag_lock()  cyg_mutex_lock(&dpMutex)
#define diag_unlock() cyg_mutex_unlock(&dpMutex)
#else
#define diag_printf_m(fmt, args...) diag_printf(fmt, ##args)
#define diag_init()  /* noop */
#define diag_lock()  /* noop */
#define diag_unlock() /* noop */
#endif
```

```
typedef unsigned char tStack[4096];

#define NumPorts 16
```

We use one thread per serial port plus an additional "background" thread that reports statistics. Each thread needs a stack, a thread object, and a thread handle:

```
cyg_thread echoThread[NumPorts], backgroundThread;
tStack echoStack[NumPorts], backgroundStack;
cyg_handle_t echoHandle[NumPorts], backgroundHandle;
cyg_thread_entry_t echoTask, backgroundTask;


// Here is where user execution starts

void cyg_user_start(void)
{
  int i;
  diag_printf("Entering cyg_user_start() function\n");

  diag_init();
```

For each of the serial ports, create and initialize a thread whose entry point is the "**echoTask**" function:

```
  for (i=0; i<NumPorts; ++i)
    {
      setInterfaceType(i,InterfaceSelect232);
      cyg_thread_create(4, echoTask, i,
                        "echo thread", echoStack[i], sizeof echoStack[i],
                        &echoHandle[i],&echoThread[i]);
      cyg_thread_resume(echoHandle[i]);
    }
```

Create and initialize the background thread:

```
  cyg_thread_create(5, backgroundTask, (cyg_addrword_t) -1,
    "background thread", backgroundStack, sizeof backgroundStack,
                    &backgroundHandle,&backgroundThread);
  cyg_thread_resume(backgroundHandle);

  // returning from this function starts scheduler
}


void done(void)
{
  for (;;)
    ;
}
```

```
/* per-port count of bytes echoed */

unsigned totalBytes[NumPorts];

/* this is a simple thread that echos data on a serial port */

void echoTask(cyg_addrword_t data)
{
  unsigned char buf[512];
  int portNum = (int)data;
  cyg_uint32 len;
  Cyg_ErrNo status;
  cyg_io_handle_t ioHandle;
  cyg_serial_info_t serConfig;
  char devName[32];

  diag_printf_m("%d: Beginning execution\n",portNum);
```

The **nserial** device names are "**/dev/ser0**" through "**/dev/ser15**". The "**data**" parameter to the **echoTask** tells us which port to handle.

```
  sprintf(devName,"/dev/ser%d",portNum);

  status = cyg_io_lookup(devName, &ioHandle);

  if (status != ENOERR)
    {
      diag_printf_m("%d: ERROR, cyg_io_lookup returned %d:
                                %s\n",portNum,status,strerror(status));
      done();
    }
```

Use cyg_io_get_config() to read the current serial port configuration structure:

```
  len = sizeof serConfig;
  status = cyg_io_get_config(ioHandle, CYG_IO_GET_CONFIG_SERIAL_INFO, &serConfig, &len);

  if (status != ENOERR)
    {
      diag_printf_m("%d: ERROR, cyg_io_get_config returned %d:
                              %s\n",portNum,status,strerror(status));
      done();
    }
```

Modify the port config structure to reflect the confguration we want: 38400,8,n,1 with no flow control:

```
serConfig.flags &= ~CYG_SERIAL_FLAGS_RTSCTS;
serConfig.flags &= ~CYG_SERIAL_FLAGS_XONXOFF_RXFLOW;
serConfig.flags &= ~CYG_SERIAL_FLAGS_XONXOFF_TXFLOW;

serConfig.flags = CYG_SERIAL_FLAGS_RD_SEMIBLOCK;

serConfig.baud = 38400;
serConfig.stop = CYGNUM_SERIAL_STOP_1;
serConfig.parity = CYGNUM_SERIAL_PARITY_NONE;
serConfig.word_length = CYGNUM_SERIAL_WORD_LENGTH_8;
serConfig.interface_mode = InterfaceSelect232;
```

Use cyg_io_set_config() to "write" the configuration back to the serial port:

```
len = sizeof serConfig;
status = cyg_io_set_config(ioHandle, CYG_IO_SET_CONFIG_SERIAL_INFO,
                                                    &serConfig, &len);


if (status != ENOERR)
  {
    diag_printf_m("%d: ERROR, cyg_io_get_config returned %d:
                          %s\n",portNum,status,strerror(status));
    done();
  }
```

Now just sit in a loop waiting for data to arrive and writing it back out when it does. We've placed the port in semi-blocking read mode, so calls to cyg_io_read() won't return until we've got data, but it won't wait for our buffer to be full, so remember to check the returned read length:

```
for (;;)
  {
    len = sizeof buf;
    status = cyg_io_read(ioHandle, buf, &len);

    // diag_printf_m("%d[%d]\n",portNum,len);

    // semi-blocking read shouldn't return with len of 0,
    // but let's check just the same

    if (len == 0)
      continue;

    totalBytes[data] += len;

    if (status != ENOERR && status != -EAGAIN)
      {
        diag_printf("%d: ERROR, cyg_io_read returned %d:
                              %s\n",portNum,status,strerror(status));
        done();
      }
```

```
      status = cyg_io_write(ioHandle, buf, &len);

      if (status != ENOERR)
        {
          diag_printf("%d: ERROR, cyg_io_write returned %d:
                                %s\n",portNum,status,strerror(status));
          done();
        }
    }
}
```

Background task prints out status message once per second that contains the total bytes echoed on each port and the number of idle-loops/sec reported by eCos (a rough measure of CPU utilization):

```
// idle thread loop count provided by eCos
extern volatile unsigned idle_thread_loops;

void backgroundTask(cyg_addrword_t data)
{
  int i;
  unsigned lastIdleThreadLoops, thisIdleThreadLoops;

  lastIdleThreadLoops = idle_thread_loops;

  while (1)
    {
      cyg_thread_delay(100);  // 1 second
      thisIdleThreadLoops = idle_thread_loops;

      diag_lock();
      diag_printf("Bytes transfered: ");
      for (i=0; i<NumPorts; ++i)
        diag_printf("%d ",totalBytes[i]);
      diag_printf(" -- Idle Loops/Sec:
                               %d\n",thisIdleThreadLoops - lastIdleThreadLoops);
      diag_unlock();

      lastIdleThreadLoops = thisIdleThreadLoops;
    }
}
```

Trademark Notices

Comtrol and DeviceMaster are trademarks of Comtrol Corporation. Other product names mentioned herein may be trademarks and/or registered trademarks of their respective owners.

Second Edition, June 25, 2004

Comtrol Corporation makes no representations or warranties with regard to the contents of this document or to the suitability of the Comtrol product for any particular purpose. Specifications subject to change without notice. Some software or features may not be available at the time of publication. Contact your reseller for current product information.

Document Number: 2000241 Rev. B