**DYNAMIC ENGINEERING**
150 Dubois St. C  Santa Cruz, CA  95060
831-457-8891  **Fax** 831-457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

# PmcSer5, SerScc and SerUart Driver Documentation

Manual Revision A
Corresponding Hardware: Revision C
Fab Number 10-2003-0303
PROM revision A

**PmcSer5**
WDM drivers for the PMC-Serial-RTN5
Serial Data Interface
PMC Module

Dynamic Engineering
150 Dubois, Suite C
Santa Cruz, CA 95060
831 457 8891
FAX 831 457 4793


Manual Revision A. Revised July 18, 2008.

# Table of Contents

## Introduction

The PmcSer5, SerUart and SerScc drivers are Win32 driver model (WDM) device drivers for the PMC-Serial-RTN5 from Dynamic Engineering. The PMC-Serial-RTN5 board has an XR16C854 Quad UART and a Z85230 Enhanced Serial Communication Controller. A Spartan2-200 Xilinx FPGA implements the PCI interface, protocol control and status, and device interfaces for four UART channels and two SCC channels.

When the PMC-Serial-RTN5 is recognized by the PCI bus configuration utility it will start the PmcSer5 driver. The PmcSer5 driver enumerates the UART and SCC channels and creates separate device object for each. This allows the I/O channels to be totally independent while the base driver controls the device items that are common to all the channels. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of data in and out of the I/O channel devices.

## Note

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PMC-Serial-RTN5 user manual (also referred to as the hardware manual).

## Driver Installation

There are several files provided in each driver package. These files include PmcSer5.sys, SerUart.sys, SerScc.sys, PmcSer.inf, SerChan.inf, DDPmcSer5.h, DDSerUart.h, DDSerScc.h, PmcSer5GUID.h, SerUartGUID.h, SerSccGUID.h, PmcSer5Test.exe, and PmcSer5Test source files.

DDPmcSer5.h, DDSerUart.h and DDSerScc.h are C header files that define the Application Program Interface (API) to the drivers. PmcSer5GUID.h, SerUartGUID.h and SerSccGUID.h are C header files that define the device interface identifiers for the PmcSer5, SerUart and SerScc drivers. These files are required at compile time by any application that wishes to interface with the drivers, but they are not needed for driver installation.

PmcSer5Test.exe is a sample Win32 console application that makes calls into the drivers to test each driver call without actually writing any application code. It is not required during the driver installation.

To run PmcSer5Test.exe, open a command prompt console window and type ***PmcSer5Test -d0 -?, PmcSer5Test -u0 -?*** or ***PmcSer5Test -s0 -?***. This will display a list of commands for the respective driver (the PmcSer5Test.exe file must be in the directory that the window is referencing). The commands are all of the form

***PmcSer5Test -dn -im*** where ***n*** and ***m*** are the device number and PmcSer5 driver ioctl number respectively or ***PmcSer5Test -un -im*** where ***n*** and ***m*** are the UART channel number and SerUart driver ioctl number respectively or ***PmcSer5Test -sn -im*** where ***n*** and ***m*** are the SCC channel number and SerScc driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

## Windows 2000 Installation

Copy PmcSer5.sys, SerUart.sys, SerScc.sys, PmcSer.inf and SerChan.inf to a floppy disk, or CD if preferred.

With the PMC-Serial-RTN5 hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Select ***Next***.
_ Select ***Search for a suitable driver for my device.***
_ Select ***Next***.
_ Insert the disk prepared above in the desired drive.
_ Select the appropriate drive e.g. ***Floppy disk drives***.
_ Select ***Next***.
_ The wizard should find the PmcSer.inf file.
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the UART and SCC channels and reopen the ***New Hardware Wizard.*** Proceed as above substituting SerChan.inf for PmcSer.inf. Repeat this for the other channels as necessary.

## Windows XP Installation

Copy PmcSer5.sys, SerUart.sys, SerScc.sys, PmcSer.inf and SerChan.inf to a floppy disk, or CD if preferred.

With the PMC-Serial-RTN5 hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Insert the disk prepared above in the desired drive.
_ Select ***No when asked to connect to Windows Update***.
_ Select ***Next***.
_ Select ***Install the software automatically.***
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the UART and SCC channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

## Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware. A handle can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system. The interface to the device is identified using globally unique identifiers (GUIDs), which are defined in PmcSer5GUID.h, SerUartGUID.h and SerSccGUID.h.

Below is example code for opening handles for device *0*.
**Note**: In order to build an application with the code below you must link with setupapi.lib and include the following header files: windows.h, stdio.h, stdlib.h, objbase.h, initguid.h, setupapi.h, winerror.h, winioctl.h, process.h and memory.h

```
// Maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Handles to device objects
HANDLE hPmcSer5             =  INVALID_HANDLE_VALUE;
HANDLE hSerScc[NUM_SCHAN]   = {INVALID_HANDLE_VALUE, INVALID_HANDLE_VALUE};
HANDLE hSerSccOL[NUM_SCHAN] = {INVALID_HANDLE_VALUE, INVALID_HANDLE_VALUE};
HANDLE hSerUart[NUM_UCHAN]  = {INVALID_HANDLE_VALUE, INVALID_HANDLE_VALUE,
                               INVALID_HANDLE_VALUE, INVALID_HANDLE_VALUE};
// PmcSer5 channel number
UCHAR                           chan;
// Return status from command
LONG                            status;
// Handle to device information structure for the interface to devices
HDEVINFO                        hDeviceInfo;
// The actual symbolic link name to use in the createfile
CHAR                            deviceName[MAX_DEVICE_NAME];
// Size of buffer reguired to get the symbolic link name
DWORD                           requiredSize;
// Interface data for this device
SP_DEVICE_INTERFACE_DATA        interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

hDeviceInfo = SetupDiGetClassDevs(
                    (LPGUID)&GUID_DEVINTERFACE_PMC_SER5,
                            NULL,
                            NULL,
                            DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

   interfaceData.cbSize = sizeof(interfaceData);
```

```
// Find the interface for device 0
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                          (LPGUID)&GUID_DEVINTERFACE_PMC_SER5,
                                0,
                                &interfaceData))
{
   status = GetLastError();
   if(status == ERROR_NO_MORE_ITEMS)
   {
      printf("**Error: couldn't find device(no more items), (%d)\n", 0);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
   else
   {
      printf("**Error: couldn't enum device, (%d)\n", status);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Found our device, get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    NULL,
                                    0,
                                    &requiredSize,
                                    NULL))
{
   if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
   {
      printf("**Error: couldn't get interface detail, (%d)\n",
              GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
if(pDeviceDetail == NULL)
{
   printf("**Error: couldn't allocate interface detail\n");
   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   exit(-1);
}
pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
```

```
// Get the detail info
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    pDeviceDetail,
                                    requiredSize,
                                    NULL,
                                    NULL))
{
   printf("**Error: couldn't get interface detail(2), (%d)\n",
          GetLastError());

   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   free(pDeviceDetail);
   exit(-1);
}

// Save the name
lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);
SetupDiDestroyDeviceInfoList(hDeviceInfo);

// Open driver - Create the handle to the device
hPmcSer5 = CreateFile(deviceName,
                      GENERIC_READ    | GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL, OPEN_EXISTING, NULL, NULL);

if(hPmcSer5 == INVALID_HANDLE_VALUE)
{
   printf("**Error: couldn't open %s, (%d)\n", deviceName, GetLastError());
   exit(-1);
}

hDeviceInfo = SetupDiGetClassDevs(
                  (LPGUID)&GUID_DEVINTERFACE_SER_SCC,
                          NULL,
                          NULL,
                          DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);
```

```
for(chan = 0; chan < NUM_SCHAN; chan++)
{// Find the interface for chan device
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                             (LPGUID)&GUID_DEVINTERFACE_SER_SCC,
                                   chan,
                                   &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n",
                 chan);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
      else
      {
         printf("**Error: couldn't enum device, (%d)\n", status);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }

 // Found our device, get the details data to obtain the symbolic link name
   if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                       &interfaceData,
                                       NULL,
                                       0,
                                       &requiredSize,
                                       NULL))
   {
      if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
      {
         printf("**Error: couldn't get interface detail, (%d)\n",
                GetLastError());

         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }

 // Allocate a buffer to get detail
   pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
   if(pDeviceDetail == NULL)
   {
      printf("**Error: couldn't allocate interface detail\n");
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
   pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
```

```
// Get the detail info
  if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                      &interfaceData,
                                      pDeviceDetail,
                                      requiredSize,
                                      NULL,
                                      NULL))
  {
     printf("**Error: couldn't get interface detail(2), (%d)\n",
             GetLastError());

     SetupDiDestroyDeviceInfoList(hDeviceInfo);
     free(pDeviceDetail);
     exit(-1);
  }

// Save the name
  lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
  free(pDeviceDetail);

// Open driver - Create the handle to the device
  hSerScc[chan] = CreateFile(deviceName,
                             GENERIC_READ    | GENERIC_WRITE,
                             FILE_SHARE_READ | FILE_SHARE_WRITE,
                             NULL,
                             OPEN_EXISTING,
                             NULL,
                             NULL);

  if(hSerScc[chan] == INVALID_HANDLE_VALUE)
  {
     printf("**Error: couldn't open %s, (%d)\n",
             deviceName, GetLastError());

     SetupDiDestroyDeviceInfoList(hDeviceInfo);
     exit(-1);
  }

  hSerSccOL[chan] = CreateFile(deviceName,
                               GENERIC_READ    | GENERIC_WRITE,
                               FILE_SHARE_READ | FILE_SHARE_WRITE,
                               NULL,
                               OPEN_EXISTING,
                               FILE_FLAG_OVERLAPPED,
                               NULL);
```

```
   if(hSerSccOL[chan] == INVALID_HANDLE_VALUE)
   {
      printf("**Error: couldn't open %s, (%d)\n",
              deviceName, GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

SetupDiDestroyDeviceInfoList(hDeviceInfo);

hDeviceInfo = SetupDiGetClassDevs(
              (LPGUID)&GUID_DEVINTERFACE_SER_UART,
                      NULL,
                      NULL,
                      DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

for(chan = 0; chan < NUM_UCHAN; chan++)
{// Find the interface for chan device
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_SER_UART,
                                   chan,
                                   &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n",
                 chan);

         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
      else
      {
         printf("**Error: couldn't enum device, (%d)\n", status);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }
```

```
// Found our device, get the details data to obtain the symbolic link name
  if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                      &interfaceData,
                                      NULL,
                                      0,
                                      &requiredSize,
                                      NULL))
  {
     if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
     {
        printf("**Error: couldn't get interface detail, (%d)\n",
                GetLastError());

        SetupDiDestroyDeviceInfoList(hDeviceInfo);
        exit(-1);
     }
  }

// Allocate a buffer to get detail
  pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
  if(pDeviceDetail == NULL)
  {
     printf("**Error: couldn't allocate interface detail\n");
     SetupDiDestroyDeviceInfoList(hDeviceInfo);
     exit(-1);
  }
  pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Get the detail info
  if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                      &interfaceData,
                                      pDeviceDetail,
                                      requiredSize,
                                      NULL,
                                      NULL))
  {
     printf("**Error: couldn't get interface detail(2), (%d)\n",
             GetLastError());

     SetupDiDestroyDeviceInfoList(hDeviceInfo);
     free(pDeviceDetail);
     exit(-1);
  }

// Save the name
  lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
  free(pDeviceDetail);
```

```
 // Open driver - Create the handle to the device
   hSerUart[chan] = CreateFile(deviceName,
                               GENERIC_READ    | GENERIC_WRITE,
                               FILE_SHARE_READ | FILE_SHARE_WRITE,
                               NULL,
                               OPEN_EXISTING,
                               NULL,
                               NULL);

   if(hSerUart[chan] == INVALID_HANDLE_VALUE)
   {
      printf("**Error: couldn't open %s, (%d)\n",
              deviceName, GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

SetupDiDestroyDeviceInfoList(hDeviceInfo);
```

## IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (see below), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE        hDevice,          // Handle opened with CreateFile()
  DWORD         dwIoControlCode, // Control code defined in API header file
  LPVOID        lpInBuffer,       // Pointer to input parameter
  DWORD         nInBufferSize,    // Size of input parameter
  LPVOID        lpOutBuffer,      // Pointer to output parameter
  DWORD         nOutBufferSize,   // Size of output parameter
  LPDWORD       lpBytesReturned, // Pointer to return length parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to overlapped structure
);                                //   used for asynchronous I/O
```

**The IOCTLs defined for the PmcSer5 driver are described below:**

### IOCTL_PMC_SER5_GET_INFO

***Function:*** Returns the current driver version, user switch value, UART device ID and revision, and Xilinx revision.
***Input:*** None
***Output:*** PMC_SER5_DRIVER_DEVICE_INFO structure
***Notes:*** This call only accesses the hardware to read the user-switch setting. All other values are constants or are read and stored during driver start-up. See DDPmcSer5.h for the definition of PMC_SER5_DRIVER_DEVICE_INFO.

**IOCTL_PMC_SER5_SET_BASE_CONFIG**

***Function:*** Sets IO configuration parameters in the PMC-Serial-RTN5 base control register.
***Input:*** PMC_SER5_BASE_CONFIG structure
***Output:*** None
***Notes:*** Selects the reference clock source(s) for the UART and SCC devices, the bus timeout interrupt enable state, and other miscellaneous controls. This call controls the routing of the SCC bi-directional signals Sync and TRxClk to either input or output drivers and which input drives the SCC DCD and RTxClk lines. These driver controls are summarized in the table below.

| Structure field | SCC Signal | When True | When False |
|---|---|---|---|
| SRTxClkA | /RTxCA | driven by IO_1P | driven by IO_5P |
| SRTxClkB | /RTxCB | driven by IO_9P | driven by IO_13P |
| SDcdA | /DCDA | driven by IO_0P | driven by IO_4N |
| SDcdB | /DCDB | driven by IO_8P | driven by IO_12N |
| STRxClkAin | /TRxCA | driven by IO_5N | drives IO_7N |
| STRxClkBin | /TRxCB | driven by IO_13N | drives IO_15N |
| SSyncAin | /SYNCA | driven by IO_4P | drives IO_6P and AUXOUT0 |
| SSyncBin | /SYNCB | driven by IO_12P | drives IO_14P and AUXOUT1 |

SInvRtsA , when true, inverts the polarity of the RTS signal from channel A.
UFFStatEn, when true, enables the status of the eight UART FIFOs (two per channel) on the UART data-bus in place of UART read data regardless of the state of the address lines.

RxATerm and RxBTerm, when true, enables the 100Ω shunt termination across the differential pairs of the respective UART receivers. (channels C and D are RS-232)

The direction of the SYNC and TRxC signals to/from the SCC must be set independently using the SET_SCC_CLOCK_CONFIG and SET_SCC_SYNC_CONFIG calls. See DDPmcSer5.h for the definition of PMC_SER5_BASE_CONFIG. See DDSerScc.h for information on the sync and clock configuration commands. See the PMC-Serial-RTN5 user manual for more information on the IO pin-outs.

**IOCTL_PMC_SER5_GET_BASE_CONFIG**

***Function:*** Returns the configuration of the base control register.
***Input:*** None
***Output:*** PMC_SER5_BASE_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_PMC_SER5_SET_TIMEOUT_CONFIG

***Function:*** Sets the bus timeout count and data value.
***Input:*** PMC_SER5_TIMEOUT_CONFIG structure
***Output:*** None
***Notes:*** Sets the timeout count, the number of PCI clocks that the bus interface will wait before signaling a timeout interrupt and returning the data specified in the timeout data field. This will only occur when UART pre-read data is accessed and there is insufficient data stored to satisfy the request. See DDPmcSer5.h for the definition of PMC_SER5_TIMEOUT_CONFIG.

### IOCTL_PMC_SER5_GET_TIMEOUT_CONFIG

***Function:*** Returns the bus timeout count and data values.
***Input:*** None
***Output:*** PMC_SER5_TIMEOUT_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_PMC_SER5_GET_STATUS

***Function:*** Returns the status bits in the INT_STAT register.
***Input:*** None
***Output:*** Unsigned long integer
***Notes:*** Reads and returns the value of the INT_STAT register which indicates the state of the various interrupt sources. This call also clears all the latched UART and SCC interrupt bits as well as the latched timer interrupt bit. See the bit definitions in the DDPmcSer5.h header file for more information.

### IOCTL_PMC_SER5_SET_ALT232_DATA_CONFIG

***Function:*** Writes enables and data values for the 24 RS-232 and 2 TTL outputs.
***Input:*** PMC_SER5_ALT232DAT_CONFIG structure
***Output:*** None
***Notes:*** If an enable for a particular bit is set to a one, the corresponding data value for that bit supersedes the previously assigned output signal. There are 24 RS-232 signals controlled by the low 24 bits and the next two bits control the two TTL AUX outputs. See the DDPmcSer5.h header file for the definition of PMC_SER5_ALT232DAT_CONFIG.

### IOCTL_PMC_SER5_GET_ALT232_DATA_CONFIG

***Function:*** Returns the alternate data values and enables for the 24 RS-232 and 2 TTL outputs.
***Input:*** None
***Output:*** PMC_SER5_ALT232DAT_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_PMC_SER5_RS232_DATA_RDBK

***Function:*** Reads the 24 RS-232 and two auxilliary input data values.
***Input:*** None
***Output:*** unsigned long integer
***Notes:*** Returns the value of the AUX/RS-232 data input bus.

### IOCTL_PMC_SER5_REGISTER_EVENT

***Function:*** Registers an event to be signaled when an interrupt occurs.
***Input:*** Handle to Event object
***Output:*** None
***Notes:*** The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when an interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. In order to un-register the event, set the event handle to NULL while making this call. The event itself must be freed separately using the CloseHandle() call, when it is no longer needed.

### IOCTL_PMC_SER5_FORCE_INTERRUPT

***Function:*** Causes a system interrupt to occur.
***Input:*** None
***Output:*** None
***Notes:*** Causes an interrupt to be asserted on the PCI bus. This IOCTL is used for development, to test interrupt processing.

### IOCTL_PMC_SER5_GET_ISR_STATUS

***Function:*** Returns the interrupt status value read in the last ISR.
***Input:*** None
***Output:*** Unsigned long integer
***Notes:*** The status contains the contents of the INT_STAT register read in the last driver interrupt service routine.

**The IOCTLs defined for the SerUart driver are described below:**

### IOCTL_SER_UART_GET_INFO

***Function:*** Returns the current driver version and Instance number of the referenced UART channel. Also returns the UART device ID and revision.
***Input:*** None
***Output:*** SER_UART_DRIVER_DEVICE_INFO structure
***Notes:*** The driver version is a constant and the instance number is assigned by the system on start-up. The instance number is equal to the number of UART channels that have already been seen by the system i.e. the first channel (channel A) is instance number zero and the other channels art instance number one through three. If more than one PMC-Serial RTN5 is installed, the numbering will continue with the UART channels of the second board. The UART device ID and revision are read during driver initialization and saved for further reference. See DDSerUart.h for the SER_UART_DRIVER_DEVICE_INFO definition.

### IOCTL_SER_UART_SET_CHAN_CONFIG

***Function:*** Sets the configuration parameters for the referenced UART channel's Xilinx control register.
***Input:*** UART_CHAN_CONFIG structure
***Output:*** None
***Notes:*** Controls the PCI bus to/from UART data interface configuration. See DDSerUart.h for the definition of UART_CHAN_CONFIG.

### IOCTL_SER_UART_GET_CHAN_CONFIG

***Function:*** Returns the configuration parameters of the referenced UART channel's Xilinx control register.
***Input:*** None
***Output:*** UART_CHAN_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_UART_SET_DATA_CONFIG

***Function:*** Sets the configuration of the UART channel's data word and baud rate.
***Input:*** UART_DATA_CONFIG structure
***Output:*** None
***Notes:*** Controls the baud rate, number of data bits, number of stop bits and the parity configuration for a UART channel. This call accesses the UART channel's LCR, DLL, and DLM registers. See DDSerUart.h for the definition of UART_DATA_CONFIG. See the XR16C854 user manual for the UART internal register descriptions.

### IOCTL_SER_UART_GET_DATA_CONFIG

***Function:*** Returns the UART channel's data configuration.
***Input:*** None
***Output:*** UART_DATA_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_UART_SET_INTEN

***Function:*** Sets the possible interrupt sources for the referenced UART channel.
***Input:*** UART_INT_CONFIG structure
***Output:*** None
***Notes:*** Selects any of seven interrupt sources for a UART channel. Accesses the UART IER register. See DDSerUart.h for the definition of UART_INT_CONFIG. See the XR16C854 user manual for the UART interrupt enable register description.

### IOCTL_SER_UART_GET_INTEN

***Function:*** Returns the interrupt enable configuration for the UART channel.
***Input:*** None
***Output:*** UART_INT_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_UART_SET_MODEM_CONTROL

***Function:*** Sets the modem control signals and internal loop-back enable for the referenced UART channel.
***Input:*** UART_MODEM_CONTROL structure
***Output:*** None
***Notes:*** Controls the state of the modem control signals (RTS, DTR) and internal loop-back signals (OP1, OP2) for a UART channel. Also controls the baud rate generator pre-scale divide-by-four circuit. Accesses the UART MCR register. See DDSerUart.h for the definition of UART_MODEM_CONTROL. See the XR16C854 user manual for the UART modem control register description.

### IOCTL_SER_UART_GET_MODEM_CONTROL

***Function:*** Returns the modem control signals for the UART channel.
***Input:*** None
***Output:*** UART_MODEM_CONTROL structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_UART_SET_FLOW_CONTROL_PARAMS

***Function:*** Sets the flow control parameters for the UART channel.
***Input:*** UART_FLOW_PARAMS structure
***Output:*** None
***Notes:*** Sets the Rx and Tx FIFO trigger levels, Rx hysteresis value, and the Xon and Xoff character values. Accesses the UART FCTR, EMSR, TRG, XON1, XON2, XOFF1, and XOFF2 registers. See DDSerUart.h for the definition of UART_FLOW_PARAMS. See the XR16C854 user manual for the UART internal register descriptions. The EMSR and TRG registers are write only, so there is no corresponding GET_UART_FLOW_CONTROL_PARAMS call.

### IOCTL_SER_UART_SET_FLOW_CONTROL_MODE

***Function:*** Sets the flow control mode for the UART channel.
***Input:*** UART_FLOW_CONFIG structure
***Output:*** None
***Notes:*** Controls whether hardware, software, or no flow control is used for a UART channel, and further details of the selected mode. Accesses the UART EFR register. See DDSerUart.h for the definition of UART_FLOW_CONFIG. See the XR16C854 user manual for the UART enhanced function register description.

### IOCTL_SER_UART_GET_FLOW_CONTROL_MODE

***Function:*** Returns the flow control mode for the UART channel.
***Input:*** None
***Output:*** UART_FLOW_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_UART_CONFIGURE_FIFOS

***Function:*** Enables and/or resets Rx, Tx or both of the UARTchannel's FIFOs.
***Input:*** UART_FIFO_CONTROL enumeration type
***Output:*** None
***Notes:*** Controls whether the UART FIFOs are enabled or disabled. If the FIFOs are enabled; either the transmit, receive or both FIFOs can be reset. See DDSerUart.h for the definition of UART_FIFO_ CONTROL. An Rx FIFO reset will also delete any data pre-read from the Rx FIFO.

### IOCTL_SER_UART_GET_STATUS

***Function:*** Reads various status values for the UART channel.
***Input:*** None
***Output:*** UART_STATUS structure
***Notes:*** Reads and returns the value of the UART channel's interrupt status register, line status register, and modem status register as well as the Rx and Tx FIFO data counts. See DDSerUart.h for the definition of UART_STATUS.

### IOCTL_SER_UART_SET_TIME_OUT

***Function:*** Sets the I/O Timeout value.
***Input:*** Timeout in milliseconds (unsigned long integer)
***Output:*** None
***Notes:*** Sets the time the driver will wait for an IO request to complete (read or write). If the value is set to zero (reset value), the wait will be infinite. This timeout is set once when the read or write is started, so the value should exceed the expected data transfer time for the requested buffer size.

### IOCTL_SER_UART_GET_INTSTAT

***Function:*** Returns the status bits in the INT_STAT register.
***Input:*** None
***Output:*** Unsigned long integer
***Notes:*** Reads and returns the value of the INT_STAT register which indicates the state of the various interrupt sources. This call also clears the latched status for this UART channel and the latched timer interrupt bit. See the bit definitions in the DDSerUart.h header file for more information.

### IOCTL_SER_UART_REGISTER_EVENT

***Function:*** Registers an event to be signaled when a user interrupt occurs.
***Input:*** Handle to Event object
***Output:*** None
***Notes:*** The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when an interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. In order to un-register the event, set the event handle to NULL while making this call. The event itself must be freed separately using the CloseHandle() call, when it is no longer needed.

### IOCTL_SER_UART_ENABLE_INTERRUPT

***Function:*** Sets the user interrupt enable to true for the channel referenced.
***Input:*** None
***Output:*** None
***Notes:*** This call sets the UART channel control interrupt enable, leaving all other bit values in the channel control register the same. This IOCTL is used when user interrupt processing is initiated and in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. The read and write interrupts operate independently from this enable, under driver control.

### IOCTL_SER_UART_DISABLE_INTERRUPT

***Function:*** Clears the user interrupt enable for the referenced UART channel.
***Input:*** None
***Output:*** None
***Notes:*** Clears the interrupt enable for the UART channel referenced. This IOCTL is used when interrupt processing is no longer desired.

### IOCTL_SER_UART_GET_ISR_STATUS

***Function:*** Returns the interrupt status and the relevant UART interrupt status register value read in the last ISR.
***Input:*** None
***Output:*** SER_UART_INT_STAT structure
***Notes:*** The status contains the contents of the INT_STAT register read in the last driver interrupt service routine execution and the interrupt status register value for the referenced UART channel read in the last ISR. See DDSerUart.h for the definition of SER_UART_INT_STAT.

### IOCTL_SER_UART_SET_EXPECTED_BAUDRATE

***Function:*** Sets an incremental timeout for received UART data in order to detect when the transfer has finished. Useful when the requested data buffer-size is greater than the actual amount of data received.
***Input:*** Baud rate in bits per second (unsigned long integer)
***Output:*** None
***Notes:*** Used to calculate the time the driver will wait for each receive interrupt while a ReadFile call is in progress. When the read is started and each time the ISR runs for the received data available interrupt, the timeout is reinitialized to this value (a larger value is used when FIFOs are enabled). Once the timeout has expired (provided the buffer length requested has not been satisfied), any data left in the receive FIFO will be read and the call will return with STATUS_SUCCESS. If the value is set to zero (default), the wait will be infinite. This timeout is only used for read/receptions and will supersede the timeout value entered in the IOCTL_SER_UART_SET_TIME_OUT call.

**The IOCTLs defined for the SerScc driver are described below:**

### IOCTL_SER_SCC_GET_INFO

***Function:*** Returns the current driver version and Instance number of the referenced SCC channel.
***Input:*** None
***Output:*** SER_SCC_DRIVER_DEVICE_INFO structure
***Notes:*** This call does not access the hardware. The driver version is a constant and the instance number is assigned by the system on start-up. The instance number is equal to the number of SCC channels that have already been seen by the system i.e. the first channel (channel A) is instance number zero and the second is instance number one. If more than one PMC-Serial RTN5 is installed, the numbering will continue with the SCC channels of the second board. See DDSerScc.h for the definition of SER_SCC_DRIVER_DEVICE_INFO.

### IOCTL_SER_SCC_SET_CLOCK_CONFIG

***Function:*** Sets the clock source and time constant for the baud-rate generator, the Tx, Rx clock sources, and the clock multiple value. Also determines the direction and source of the TRxClk signal.
***Input:*** SCC_CLOCK_CONFIG structure
***Output:*** None
***Notes:*** The baud rate is determined by the following formula:
Osc freq/(2 * clock multiple * (BaudDiv + 2)). See DDSerScc.h for the definition of SCC_CLOCK_CONFIG.

### IOCTL_SER_SCC_GET_CLOCK_CONFIG

***Function:*** Returns the SCC channel's Tx and Rx clock source, rate, etc.
***Input:*** None
***Output:*** SCC_CLOCK_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_SCC_SET_DATA_CONFIG

***Function:*** Sets the SCC channel's Tx and Rx data word size, parity, and encoding.
***Input:*** SCC_DATA_CONFIG structure
***Output:*** None
***Notes:*** The Rx data size can be 5, 6, 7, or 8 bits. Transmit data sizes less than five bits are possible, but require that the data be pre-formatted before being written to the transmit data buffer. See the SCC user manual for the details of this process. The number of stop bits can be 1, 1.5, 2, or 0. If zero stop bits are selected, this enables synchronous mode. If the 1x clock multiplier is selected, the 1.5 stop bit selection is not allowed. See DDSerScc.h for the definition of SCC_DATA_CONFIG.

### IOCTL_SER_SCC_GET_DATA_CONFIG

***Function:*** Returns the SCC channel's Tx and Rx data word size, parity, and encoding.
***Input:*** None
***Output:*** SCC_DATA_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_SCC_SET_SYNC_CONFIG

***Function:*** Sets the SCC channel's CRC parameters sync patterns, and sync type.
***Input:*** SCC_SYNC_CONFIG structure
***Output:*** None
***Notes:*** In mono-sync mode, Sync0 contains the transmit sync and Sync1 contains the receive sync. In bi-sync mode, the sync character is contained in both fields with the lower bits in Sync0. In all cases the values are right justified. In SDLC mode, the SDLC flag is loaded automatically and Sync0 contains the secondary address field to compare against the address field of the SDLC frame. This process is modified if SyncNoLd is true in the SCC_RX_CONFIG, in this case only the upper four address bits are compared, so the receiver will respond to a range of 16 addresses. If external sync mode is selected, the direction of the sync signal is automatically changed to an input. The base control register must be configured accordingly. See the SCC user manual for more information on the various sync modes. See DDSerScc.h for the definition of SCC_SYNC_CONFIG.

### IOCTL_SER_SCC_GET_SYNC_CONFIG

***Function:*** Returns the SCC channel's CRC parameters and sync type.
***Input:*** None
***Output:*** SCC_SYNC_CONFIG structure
***Notes:*** Returns the values set in the previous call except the sync pattern values.

### IOCTL_SER_SCC_SET_INT_CONFIG

***Function:*** Sets the SCC channel's interrupt configuration.
***Input:*** SCC_INT_CONFIG structure
***Output:*** None
***Notes:*** The interrupts of the SCC are divided into three groups. The receive interrupt group consists of the receive character available and special condition interrupts. The special conditions include overrun, framing error, end-of-frame (SDLC), and (if enabled) parity error. The transmit buffer empty is the only transmitter interrupt. Finally the external interrupts are Tx underrun, break/abort, sync/hunt, CTS, DCD, and baud-rate-generator count-down to zero. See the SCC user manual for more information on the interrupt behavior and see DDSerScc.h for the definition of SCC_INT_CONFIG.

### IOCTL_SER_SCC_GET_INT_CONFIG

***Function:*** Returns the SCC channel's interrupt configuration.
***Input:*** None
***Output:*** SCC_INT_CONFIG structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_SER_SCC_RESET

***Function:*** Resets the referenced SCC channel.
***Input:*** None
***Output:*** None
***Notes:*** After the reset, the configuration values for the affected channel(s) are restored except that the transmitter and receiver will be stopped.

### IOCTL_SER_SCC_MISC_CMD

***Function:*** Issues the SCC channel's DPLL, CRC, and latch-reset commands. Also controls internal loop-back, auto-echo, and the SCLC status FIFO enable.
***Input:*** SCC_MISC_CMD structure
***Output:*** None
***Notes:*** See the SCC user manual for information on the various commands issued. See DDSerScc.h for the definition of SCC_MISC_CMD.

### IOCTL_SER_SCC_INIT_RX

***Function:*** Initializes the SCC channel's receiver in a particular mode.
***Input:*** SCC_RX_CONFIG structure
***Output:*** None
***Notes:*** See the SCC user manual for information on the various receive modes. See DDSerScc.h for the definition of SCC_RX_CONFIG.

### IOCTL_SER_SCC_INIT_TX

***Function:*** Initializes the SCC channel's transmitter in a particular mode.
***Input:*** SCC_TX_CONFIG structure
***Output:*** None
***Notes:*** See the SCC user manual for information on the various transmit modes and features. See DDSerScc.h for the definition of SCC_TX_CONFIG.

### IOCTL_SER_SCC_RX_EN

***Function:*** Start or stop the SCC channel's receiver.
***Input:*** enable (BOOLEAN type)
***Output:*** None
***Notes:*** When enable is set to true, the referenced receive channel is started, when enable is false the receiver is stopped.

### IOCTL_SER_SCC_TX_EN

***Function:*** Start or stop the SCC channel's transmitter.
***Input:*** enable (BOOLEAN type)
***Output:*** None
***Notes:*** When enable is set to true, the referenced transmit channel is started, when enable is false the transmitter is stopped.

### IOCTL_SER_SCC_GET_TREXT_STATUS

***Function:*** Returns the SCC channel's Tx/Rx buffer and external status
***Input:*** None
***Output:*** SCC_TREXT_STAT structure
***Notes:*** See DDSerScc.h for the definition of SCC_TREXT_STAT.

### IOCTL_SER_SCC_GET_SPEC_STATUS

***Function:*** Returns the SCC channel's special conditions status
***Input:*** None
***Output:*** SCC_SPEC_STAT structure
***Notes:*** This call returns various Special Receive Condition status bits: All Sent, Parity Error, Rx Overrun Error, CRC/Framing Error and SDLC End of Frame. The ResCode field contains the SDLC residue code. See the SCC user manual for more information on interpreting these values. See DDSerScc.h for the definition of SCC_SPEC_STAT.

### IOCTL_SER_SCC_GET_SDLC_STATUS

***Function:*** Returns the SCC channel's SDLC status SDLC frame status FIFO data and other SDLC status.
***Input:*** None
***Output:*** SCC_SDLC_STAT structure
***Notes:*** See DDSerScc.h for the definition of SCC_SDLC_STAT. Also refer to the user manual for the SCC device for a description of the SDLC function.

### IOCTL_SER_SCC_SET_TIME_OUT

***Function:*** Sets the I/O Timeout value.
***Input:*** Timeout in milliseconds (unsigned long integer)
***Output:*** None
***Notes:*** Sets the time the driver will wait for an IO request to complete (read or write). If the value is set to zero (reset value), the wait will be infinite. This timeout is set once when the read or write is started, so the value should exceed the expected data transfer time for the requested buffer size.

### IOCTL_SER_SCC_GET_INTSTAT

***Function:*** Returns the status bits in the INT_STAT register.
***Input:*** None
***Output:*** Unsigned long integer
***Notes:*** Reads and returns the value of the INT_STAT register which indicates the state of the various interrupt sources. This call also clears the latched SCC interrupt bit. See the bit definitions in the DDSerScc.h header file for more information.

### IOCTL_SER_SCC_REGISTER_EVENT

***Function:*** Registers an event to be signaled when a user interrupt occurs.
***Input:*** Handle to Event object
***Output:*** None
***Notes:*** The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when an interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. In order to un-register the event, set the event handle to NULL while making this call. The event itself must be freed separately using the CloseHandle() call, when it is no longer needed.

### IOCTL_SER_SCC_ENABLE_INTERRUPT

***Function:*** Sets the user interrupt enable to true for the SCC channel referenced.
***Input:*** None
***Output:*** None
***Notes:*** This call writes to the IEN register for the SCC channel referenced. This IOCTL is used when user interrupt processing is initiated and in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. The read and write interrupts operate independently from this enable, under driver control.

### IOCTL_SER_SCC_DISABLE_INTERRUPT

***Function:*** Clears the user interrupt enable for the referenced SCC channel.
***Input:*** None
***Output:*** None
***Notes:*** Clears the user interrupt enable for the SCC channel referenced. This IOCTL is used when interrupt processing is no longer desired.

### IOCTL_SER_SCC_GET_ISR_STATUS

***Function:*** Returns the overall interrupt status and the relevant SCC interrupt status register value read in the last ISR.
***Input:*** None
***Output:*** SER_SCC_INT_STAT structure
***Notes:*** The status contains the contents of the INT_STAT register read in the last driver interrupt service routine execution and the interrupt pending register of the SCC. See DDSerScc.h for the definition of SER_SCC_INT_STAT.

### IOCTL_SER_SCC_SET_EXPECTED_BAUDRATE

***Function:*** Sets an incremental timeout for received SCC data in order to detect when the transfer has finished. Useful when the requested data buffer-size is greater than the actual amount of data received.
***Input:*** Baud rate in bits per second (unsigned long integer)
***Output:*** None
***Notes:*** Used to calculate the time the driver will wait for each receive interrupt while a ReadFile call is in progress. When the read is started and each time the ISR runs for the received data available interrupt, the timeout is reinitialized to this value (approximately 64 bit-times). Once the timeout has expired (provided the buffer length requested has not been satisfied), any data left in the receive FIFO or buffer will be read and the call will return with STATUS_SUCCESS. If the value is set to zero (default), the wait will be infinite. This timeout is only used for read/receptions and will supersede the timeout value entered in the IOCTL_SER_SCC_SET_TIME_OUT call.

### Write

Data to be sent from the transmitter is written to the transmit FIFO using a WriteFile() call. The user supplies the device handle, a pointer to the buffer containing the data, the number of bytes to write, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. If the number of bytes requested exceeds the size of the buffer available, the driver will use interrupts to detect when more data can be written to the device. For a UART channel, if 16-bit or 32-bit writes are enabled, they will be used to implement this command. See Win32 help files for details the of the WriteFile() call.

### Read

Received data can be read from the receive FIFO using a ReadFile() call. The user supplies the device handle, a pointer to the buffer to store the data in, the number of bytes to read, a pointer to a variable to store the amount of data actually transferred, and a pointer to an optional Overlapped structure for performing asynchronous IO. If the number of bytes requested exceeds the receive FIFO size, the driver will use interrupts to detect when more data has arrived. Timeouts can be set to terminate the call when insufficient data is received. For a UART channel, if 16-bit or 32-bit reads are enabled, they will be used to implement this command. See Win32 help files for the details of the ReadFile() call.

## Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering’s products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

## Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be cockpit error rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer's making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer's invoicing policy.

### Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is $125. An open PO will be required.

## For Service Contact:

Customer Service Department
Dynamic Engineering
150 Dubois, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 fax
support@dyneng.com