

---

# TPMC812-SW-62

## Windows NT Device Driver

Sercos PMC with 2 Encoder Interfaces

## User Manual

Issue 1.0 Version 1.0.0

August 2002

---

**TEWS TECHNOLOGIES GmbH**

Am Bahnhof 7  
Phone: +49-(0)4101-4058-0  
e-mail: info@tews.com

25469 Halstenbek / Germany  
Fax: +49-(0)4101-4058-19  
www.tews.com

**TEWS TECHNOLOGIES LLC**

1 E. Liberty Street, Sixth Floor  
Phone: +1 (775) 686 6077  
e-mail: usasales@tews.com

Reno, Nevada 89504 / USA  
Fax: +1 (775) 686 6024  
www.tews.com

**TPMC812-SW-62**

Sercos PMC with 2 Encoder Interfaces

Windows NT Device Driver

This document contains information, which is proprietary to TEWS TECHNOLOGIES GmbH. Any reproduction without written permission is forbidden.

TEWS TECHNOLOGIES GmbH has made any effort to ensure that this manual is accurate and complete. However TEWS TECHNOLOGIES GmbH reserves the right to change the product described in this document at any time without notice.

TEWS TECHNOLOGIES GmbH is not liable for any damage arising out of the application or use of the device described herein.

©2002 by TEWS TECHNOLOGIES GmbH

<b>Issue</b>	<b>Description</b>	<b>Date</b>
1.0	First Issue	August 14, 2002

# Table of Content

<b>1</b>	<b>INTRODUCTION.....</b>	<b>5</b>
<b>2</b>	<b>INSTALLATION.....</b>	<b>6</b>
	2.1 Software Installation.....	6
<b>3</b>	<b>TPMC812 DEVICE DRIVER PROGRAMMING .....</b>	<b>7</b>
	TPMC812 Files and I/O Functions .....	8
	3.1.1 Opening a TPMC812 Device .....	8
	3.1.2 Closing a TPMC812 Device .....	10
	3.1.3 TPMC812 Device I/O Control Functions.....	12
	3.1.3.1 IOCTL_TP812_WRITE_REG.....	15
	3.1.3.2 IOCTL_TP812_READ_REG .....	17
	3.1.3.3 IOCTL_TP812_WRITE_MEM .....	19
	3.1.3.4 IOCTL_TP812_READ_MEM.....	21
	3.1.3.5 IOCTL_TP812_INIT_INT1.....	23
	3.1.3.6 IOCTL_TP812_DEINIT_INT1.....	24
	3.1.3.7 IOCTL_TP812_READ_INT1 .....	25
	3.1.3.8 IOCTL_TP812_SYNCCLK_ENA.....	27
	3.1.3.9 IOCTL_TP812_SYNCCLK_DISA.....	28
	3.1.3.10 IOCTL_TP812_READ_ENC.....	29
	3.1.3.11 IOCTL_TP812_CLEAR_ENC.....	31
<b>4</b>	<b>EXAMPLE APPLICATIONS .....</b>	<b>33</b>
	4.1 Simple Example Application.....	33
	4.2 Sercos Master Example Application.....	33
	4.2.1 Sercos driver access functions .....	35
	4.2.1.1 Write_SC_sercon_reg .....	35
	4.2.1.2 Read_SC_sercon_reg.....	37
	4.2.1.3 Write_SC_sercon_mem .....	38
	4.2.1.4 Read_SC_sercon_mem .....	40
	4.2.2 User supplied functions.....	42
	4.2.2.1 SercosMasterApplicationDataInfo .....	42
	4.2.2.2 TransferSercosMasterApplicationData.....	43
	4.2.3 Sercos Library Functions .....	45
	4.2.3.1 InitSercosMaster.....	45
	4.2.3.2 StartSercosMasterPhase.....	46
	4.2.3.3 GetSercosMasterPhaseStatus .....	48
	4.2.3.4 StartSercosMasterDriveConnection .....	49
	4.2.3.5 GetSercosMasterConnectionStatus .....	50
	4.2.3.6 StartSercosMasterServiceTransfer .....	51
	4.2.3.7 AbortSercosMasterServiceTransfer .....	53
	4.2.3.8 GetSercosMasterServiceStatus .....	54
	4.2.3.9 PrepareSercosMasterForPhase3.....	56
	4.2.4 Data Structures .....	58
	4.2.4.1 TP812_SC_CB_STRUCT .....	58
	4.2.4.2 USERRINGINFO .....	60
	4.2.4.3 ATUSERDATA .....	61
	4.2.4.4 ATUSERTIME .....	62
	4.2.4.5 MDTUSERDATA .....	63
	4.2.5 Definitions (Constant Values) .....	64
	4.2.5.1 Baudrate .....	64
	4.2.5.2 State Message.....	64
	4.2.5.3 Data Direction.....	64

---

4.2.5.4	Phases.....	64
4.2.6	Startup a Ring .....	65
4.2.6.1	Error Handling .....	65
4.2.6.2	Start a new Phase .....	66
4.2.6.3	Start a Drive Connection .....	67
4.2.6.4	Start a Service transfer.....	68
4.2.7	Error Codes.....	69

---

# 1 Introduction

The TPMC812-SW-62 Windows NT device driver is a kernel mode device driver which allows the operation of the TPMC812 on an Intel or Intel-compatible x86 Windows NT 4.0 system.

The standard file and device (I/O) functions (CreateFile, CloseHandle, and DeviceIoControl) provide the basic interface for opening and closing a resource handle and for performing device I/O control operations.

The TPMC812 device driver supports the following features:

- Writing and reading of register values
- Writing and reading a buffer to/from the TPMC812 dual ported memory
- Enable and disable interrupt sources
- Read interrupts, occurred since last read
- Enable and disable synchronous master clock
- Reading encoder values
- Clearing encoder registers

The simple example software allows to make read and write accesses to the registers and to copy data buffers from or to the DPM of the TPMC812.

The sercos master example is a simple application starting the sercos-ring up to phase 4 for two drives. After starting up through the phases the speed parameter of the drives is continuously updated. (This is a special example for a system with two simulated drives.) Most systems will need some smaller adaptation to work with this example.

## **2 Installation**

The software is delivered on a 3½" HD diskette.

Following files are located on the diskette:

TPMC812.sys	Windows NT driver binary
TPMC812.h	Header file with IOCTL code definitions
TPMC812.inf	Windows NT installation script
TPMC812-SW-62.pdf	This document
\Example\*.*	Microsoft Visual C simple example application
\SercosMaster\*.*	Microsoft Visual C sercos master example application

### **2.1 Software Installation**

This section describes how to install the TPMC812 Device Driver to a Windows NT 4.0 operating systems with Intel and Intel-compatible x86 CPU.

1. Plug the TPMC812 card(s) into the system
2. Switch on your system and log on as an Administrator.
3. Insert the TPMC812 diskette into the floppy disk drive.
4. Copy all files to the desired target directory
5. Start Windows NT Explorer and look for a file called ***TPMC812.inf***
6. Right-click this file, select Install and follow the on-screen instructions.
7. Restart your computer when prompted.

The installation process copies the driver to the *%SystemRoot%\SYSTEM32\DRIVERS* directory and adds the following entries to the Registry:

***HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services\TPMC812\..***

and

***HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services\EventLog\System\TPMC812***

After successful installation the TPMC812 device driver starts automatically. The driver searches for TPMC812 modules on the PCI-bus, allocates necessary resources and creates devices (TPMC812\_1, TPMC812\_2, ...) for all found TPMC812 modules.

---

## **3 TPMC812 Device Driver Programming**

The TPMC812-SW-62 Windows NT device driver is a kernel mode device driver.

The standard file and device (I/O) functions (CreateFile, CloseHandle, and DeviceIoControl) provide the basic interface for opening and closing a resource handle and for performing device I/O control operations.

All of these standard Win32 functions are described in detail in the Windows Platform SDK Documentation (Windows base services / Hardware / Device Input and Output).

For details refer to the Win32 Programmers Reference of your used programming tools (C++, Visual Basic etc.)

## TPMC812 Files and I/O Functions

The following section doesn't contain a full description of the Win32 functions for interaction with the TPMC812 device driver. Only the required parameters are described in detail.

### 3.1.1 Opening a TPMC812 Device

Before you can perform any I/O the *TPMC812* device must be opened by invoking the **CreateFile** function. **CreateFile** returns a handle that can be used to access the *TPMC812* device.

```
HANDLE CreateFile(
    LPCTSTR lpFileName,           // pointer to name of the file
    DWORD dwDesiredAccess,       // access (read-write) mode
    DWORD dwShareMode,           // share mode
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // pointer to security attributes
    DWORD dwCreationDistribution, // how to create
    DWORD dwFlagsAndAttributes,  // file attributes
    HANDLE hTemplateFile         // handle to file with attributes to
                                // copy
);
```

#### Parameters

##### *lpFileName*

Points to a null-terminated string that specifies the name of the TPMC812 to be opened. The *lpFileName* string should be of the form `\\.\TPMC812_x` to open the device *x*. The ending *x* is a one-based number. The first device found by the driver is `\\.\TPMC812_1`, the second `\\.\TPMC812_2` and so on.

##### *dwDesiredAccess*

Specifies the type of access to the TPMC812.  
For the TPMC812 this parameter must be set to read-write access (GENERIC\_READ | GENERIC\_WRITE)

##### *dwShareMode*

Set of bit flags that specify how the object can be shared. A TPMC812 device can't be shared by several tasks. Set to 0.

##### *lpSecurityAttributes*

Pointer to a security structure. Set to NULL for TPMC812 devices.

##### *dwCreationDistribution*

Specifies which action to take on files, that exist and which action to take when files do not exist. TPMC812 devices must be always opened `OPEN_EXISTING`.

*dwFlagsAndAttributes*

Specifies the file attributes and flags for the file. This value must be set to 0 (no overlapped I/O)

*hTemplateFile*

This value must be NULL for TPMC812 devices.

**Return Value**

If the function succeeds, the return value is an open handle to the specified TPMC812 device. If the function fails, the return value is INVALID\_HANDLE\_VALUE. To get extended error information, call GetLastError.

**Example**

```
HANDLE    hDevice;

hDevice = CreateFile(
    "\\.\TPMC812_1",
    GENERIC_READ | GENERIC_WRITE,
    0,
    NULL,           // no security attrs
    OPEN_EXISTING, // TPMC812 device always open
                  // existing
    0,             // no overlapped I/O
    NULL);

if (hDevice == INVALID_HANDLE_VALUE) {
    ErrorHandler("Could not open device"); // process error
}

...
```

**See Also**

CloseHandle(), Win32 documentation CreateFile()

### 3.1.2 Closing a TPMC812 Device

The CloseHandle function closes an open TPMC812 handle.

```
BOOL CloseHandle(  
    HANDLE hDevice;                // handle to a TPMC812 device  
                                   // to close  
);
```

#### Parameters

*hDevice*

Identifies an open TPMC812 handle.

#### Return Value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

#### Example

```
HANDLE    hDevice;  
  
hDevice = CreateFile(  
    "\\.\TPMC812_1",  
    GENERIC_READ | GENERIC_WRITE,  
    0,  
    NULL,                // no security attrs  
    OPEN_EXISTING,      // TPMC812 device always open  
                        // existing  
    0,                  // no overlapped I/O  
    NULL);  
  
If (hDevice == INVALID_HANDLE_VALUE) {  
    ErrorHandler("Could not open device");    // process error  
}  
  
/* ... do some device I/O ... */  
  
if (CloseHandle(hDevice)) {  
    ErrorHandler("Could not close device"); // process error  
}  
  
...
```

## **See Also**

CreateFile(), Win32 documentation CloseHandle()

### 3.1.3 TPMC812 Device I/O Control Functions

The DeviceIoControl function sends a control code directly to a specified device driver, causing the corresponding device to perform the specified operation.

```
BOOL DeviceIoControl(
    HANDLE hDevice,                // handle to device of interest
    DWORD dwIoControlCode,        // control code of operation to
                                  // perform
    LPVOID lpInBuffer,            // pointer to buffer to supply
                                  // input data
    DWORD nInBufferSize,          // size of input buffer
    LPVOID lpOutBuffer,           // pointer to buffer to receive
                                  // output data
    DWORD nOutBufferSize,         // size of output buffer
    LPDWORD lpBytesReturned,       // pointer to variable to receive
                                  // output byte
                                  // count
    LPOVERLAPPED lpOverlapped     // pointer to overlapped structure
                                  // for asynchronous operation
);
```

#### Parameters

##### *hDevice*

Handle to the TPMC812 that is to perform the operation.

*dwIoControlCode*

Specifies the control code of the operation. This value identifies the specific operation to be performed. The following values are defined in TPMC812.h:

<b>Value</b>	<b>Meaning</b>
IOCTL_TP812_WRITE_REG	Write a value in to a register
IOCTL_TP812_READ_REG	Read a value from a register
IOCTL_TP812_WRITE_MEM	Copy the content of a buffer to the Dual Ported Memory
IOCTL_TP812_READ_MEM	Copy the content of the Dual Ported Memory in to a buffer
IOCTL_TP812_INIT_INT1	Enable interrupts
IOCTL_TP812_DEINIT_INT1	Disable interrupts
IOCTL_TP812_READ_INT1	Read which interrupts occurred since last read (poll interrupt)
IOCTL_TP812_SYNCCLK_ENA	Enable master sync clock output
IOCTL_TP812_SYNCCLK_DISA	Disable master sync clock output
IOCTL_TP812_READ_ENC	Read the actual values of the encoder channels
IOCTL_TP812_CLEAR_ENC	Clear specified encoder channel(s)

See behind for more detailed information on each control code.

**To use these TPMC812 specific control codes the header file TPMC812.h must be included in the application**

*lpInBuffer*

Pointer to a buffer that contains the data required to perform the operation.

*nInBufferSize*

Specifies the size of the buffer in bytes pointed to by lpInBuffer.

*lpOutBuffer*

Pointer to a buffer that receives the operation's output data.

*nOutBufferSize*

Specifies the size of the buffer in bytes pointed to by lpOutBuffer.

*lpBytesReturned*

Pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by lpOutBuffer. A valid pointer is required.

*lpOverlapped*

Pointer to an overlapped structure. This parameter must be set to NULL, because we do not use overlapped I/O.

## **Return Value**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

## **See Also**

Win32 documentation DeviceIoControl()

### 3.1.3.1 IOCTL\_TP812\_WRITE\_REG

This control function attempts to write a value into a register of the TPMC812 associated with the open device handle.

The needed parameters are passed in a buffer (*TP812\_IN\_WRITE\_REG\_BUF*), pointed by *lpInBuffer*, to the driver. The argument *nInBufferSize* specifies the size (size of *TP812\_IN\_WRITE\_REG\_BUF*) of the write buffer.

The *TP812\_IN\_WRITE\_REG\_BUF* structure has the following layout (see TPMC812.h):

typedef struct

```
{
    USHORT          reg_addr;
    USHORT          value;
} TP812_IN_WRITE_REG_BUF, *PTP812_IN_WRITE_REG_BUF;
```

*reg\_addr*

This argument specifies the register to write to. The register names are defined in TPMC812.H.

*value*

This argument specifies the value that will be written to the specified register.

## Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG          NumBytes;
TP812_IN_WRITE_REG_BUF iWrRegBuf;

...

// Write 0x1234 into register TP812_REG04
iWrRegBuf.reg_addr = TP812_REG04;
iWrRegBuf.value    = 0x1234;
success = DeviceIoControl(
    hCurrent,          // TPMC812 handle
    IOCTL_TP812_WRITE_REG, // control code
    &iWrRegBuf,        // buffer with control
                      // information for the driver
    sizeof(iWrRegBuf),
    NULL,             // no receive return values
    0,
    &NumBytes,        // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Register Write failed"); // process error
}

...
```

## Error Codes

STATUS_INVALID_PARAMETER	This error is returned if the size of the write buffer is too small.
--------------------------	--

## See Also

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

### 3.1.3.2 IOCTL\_TP812\_READ\_REG

This control function attempts to read the value of a register of the TPMC812 associated with the open device handle.

The needed parameters are passed in a buffer (*TP812\_IN\_READ\_REG\_BUF*), pointed by *lpInBuffer*, to the driver. The argument *nInBufferSize* specifies the size (size of *TP812\_IN\_READ\_REG\_BUF*) of the write buffer.

The return buffer (*USHORT*) must be pointed by *lpOutBuffer*. The argument *nOutBufferSize* specifies the size (size of *USHORT*) of the read buffer.

The *TP812\_IN\_READ\_REG\_BUF* structure has the following layout (see TPMC812.h):

typedef struct

```
{  
    USHORT          reg_addr;  
} TP812_IN_READ_REG_BUF, *PTP812_IN_READ_REG_BUF;
```

*reg\_addr*

This argument specifies the register to read from. The register names are defined in TPMC812.H.

## Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN        success;
ULONG          NumBytes;
TP812_IN_READ_REG_BUF  iRdRegBuf;
USHORT        usValue;

...

// Read from register TP812_VERSION
iRdRegBuf.reg_addr = TP812_VERSION;
success = DeviceIoControl(
    hCurrent,                // TPMC812 handle
    IOCTL_TP812_READ_REG,   // control code
    &iRdRegBuf,              // buffer with control
                            // information for the driver
    sizeof(iRdRegBuf),
    &usValue,               // address of return value
    sizeof(usValue),
    &NumBytes,              // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Register Read failed"); // process error
}

...
```

## Error Codes

STATUS_INVALID_PARAMETER	This error is returned if the size of the write buffer is too small.
--------------------------	--

## See Also

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

### 3.1.3.3 IOCTL\_TP812\_WRITE\_MEM

This control function attempts to write the content of a buffer into the dual ported memory register of the TPMC812 associated with the open device handle.

The needed parameters are passed in a buffer (*TP812\_IN\_WRITE\_MEM\_BUF*), pointed by *lpInBuffer*, to the driver. The argument *nInBufferSize* specifies the size (size of *TP812\_IN\_WRITE\_MEM\_BUF*) of the write buffer.

The *TP812\_IN\_WRITE\_MEM\_BUF* structure has the following layout (see TPMC812.h):

```
typedef struct
{
    USHORT          mem_offset;
    USHORT          num_words;
    USHORT          buffer[TP812_DPMBUFSIZE];
} TP812_IN_WRITE_MEM_BUF, *PTP812_IN_WRITE_MEM_BUF;
```

#### *mem\_offset*

This argument specifies the offset address of the dual ported memory the buffer will be copied to.

#### *num\_words*

This argument specifies the number of valid data words to be copied into the dual ported memory.

#### *buffer[]*

This argument specifies the buffer. The data to copy must be placed in this buffer.

## Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG           NumBytes;
TP812_IN_WRITE_MEM_BUF iWrMemBuf;

...

// Copy the buffer with 4 values to DPM-offset 0x10
iWrMemBuf.mem_offset = 0x10;
iWrMemBuf.num_words = 4;
iWrMemBuf.buffer[0] = 0x1122;
iWrMemBuf.buffer[1] = 0x3344;
iWrMemBuf.buffer[2] = 0x5566;
iWrMemBuf.buffer[3] = 0x7788;
success = DeviceIoControl(
    hCurrent, // TPMC812 handle
    IOCTL_TP812_WRITE_MEM, // control code
    &iWrMemBuf, // buffer with control
    // information for the driver
    sizeof(iWrMemBuf),
    NULL, // no receive return values
    0,
    &NumBytes, // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Memory Write failed"); // process error
}

...
```

## Error Codes

STATUS_INVALID_PARAMETER	This error is returned if the size of the write buffer is too small.
--------------------------	--

## See Also

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

### 3.1.3.4 IOCTL\_TP812\_READ\_MEM

This control function attempts to copy a part of the dual ported memory of the TPMC812 associated with the open device handle into a buffer.

The needed parameters are passed in a buffer (*TP812\_IN\_READ\_MEM\_BUF*), pointed by *lpInBuffer*, to the driver. The argument *nInBufferSize* specifies the size (size of *TP812\_IN\_READ\_MEM\_BUF*) of the write buffer.

The return buffer (array of USHORT) must be pointed by *lpOutBuffer*. The argument *nOutBufferSize* specifies the size (size of array of *USHORT*) of the read buffer.

The *TP812\_IN\_READ\_MEM\_BUF* structure has the following layout (see TPMC812.h):

```
typedef struct{
    USHORT          mem_offset;
    USHORT          num_words;
} TP812_IN_READ_MEM_BUF, *PTP812_IN_READ_MEM_BUF;
```

#### *mem\_offset*

This argument specifies the offset address of the dual ported memory the buffer will be copied from.

#### *num\_words*

This argument specifies the number of data words to be copied from the dual ported memory.

## Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG           NumBytes;
TP812_IN_READ_MEM_BUF iRdRegBuf;
USHORT         buffer[4];

...

// Copy 4 words from address 0x10 from the DPM
iRdMemBuf.mem_offset = 0x10;
iRdMemBuf.num_words = 4;
success = DeviceIoControl(
    hCurrent,                // TPMC812 handle
    IOCTL_TP812_READ_MEM,   // control code
    &iRdRegBuf,              // buffer with control
                             // information for the driver
    sizeof(iRdRegBuf),
    buffer,                  // address of return value
    sizeof(USHORT) * 4,
    &NumBytes,              // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Read Memory failed"); // process error
}

...
```

## Error Codes

STATUS_INVALID_PARAMETER	This error is returned if the size of the write buffer is too small.
--------------------------	--

## See Also

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

### 3.1.3.5 IOCTL\_TP812\_INIT\_INT1

This control function enables the interrupt source 1 globally (interrupt source 0 is not supported) of the TPMC812 associated with the open device handle. This function must be called once before any interrupts can be executed.

Because there are no parameters needed, the parameter pointer `lpInBuffer` and `lpOutBuffer` must be set to `NULL`, the size of the buffers must be set to 0.

#### Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG          NumBytes;

...

// Initialize interrupt 1
success = DeviceIoControl(
    hCurrent,          // TPMC812 handle
    IOCTL_TP812_INIT_INT1, // control code
    NULL,             // no input buffer
    0,
    NULL,            // no receive return values
    0,
    &NumBytes,      // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Initialize INT 1 failed"); // process error
}

...
```

#### See Also

Win32 documentation `DeviceIoControl()`, TPMC812 Hardware User Manual

### 3.1.3.6 IOCTL\_TP812\_DEINIT\_INT1

This control function disables the interrupt source 1 globally (interrupt source 0 is not supported) of the TPMC812 associated with the open device handle.

Because there are no parameters needed, the parameter pointer `lpInBuffer` and `lpOutBuffer` must be set to `NULL`, the size of the buffers must be set to 0.

#### Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG           NumBytes;

...

// Deinitialize interrupt 1
success = DeviceIoControl(
    hCurrent,                // TPMC812 handle
    IOCTL_TP812_INIT_DEINT1, // control code
    NULL,                    // no input buffer
    0,
    NULL,                    // no receive return values
    0,
    &NumBytes,               // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Deinitialize INT 1 failed"); // process error
}

...
```

#### See Also

Win32 documentation `DeviceIoControl()`, TPMC812 Hardware User Manual

### 3.1.3.7 IOCTL\_TP812\_READ\_INT1

This control function attempts to return a bit mask of activated interrupt sources since the last interrupt read of the TPMC812 associated with the open device handle.

There is no input parameter needed, `lpInBuffer` must be set to `NULL` and the size argument `nInBufferSize` must be set to 0.

The return buffer (ULONG) must be pointed by `lpOutBuffer`. The argument `nOutBufferSize` specifies the size (size of ULONG) of the read buffer. The interrupt sources of `TP812_REG04` are returned in the LSW and the interrupt sources of the `TP812_REG05` are returned in the MSW.

#### Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN        success;
ULONG          NumBytes;
ULONG          intSrcs;

...

// Read interrupt sources
success = DeviceIoControl(
    hCurrent,          // TPMC812 handle
    IOCTL_TP812_READ_INT1, // control code
    NULL,             // no input parameter
    0,
    &intSrcs,         // address of return value
    sizeof(intSrcs),
    &NumBytes,       // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Read interrupt failed"); // process error
}

...
```

## **Error Codes**

STATUS\_INVALID\_PARAMETER      This error is returned if the size of the write buffer is too small.

## **See Also**

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

### 3.1.3.8 IOCTL\_TP812\_SYNCCLK\_ENA

This control function enables the output of the master synchronization clock of the TPMC812 associated with the open device handle.

Because there are no parameters needed, the parameter pointer `lpInBuffer` and `lpOutBuffer` must be set to `NULL`, the size of the buffers must be set to 0.

#### Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG           NumBytes;

...

// Enable master synchronization clock
success = DeviceIoControl(
    hCurrent,                // TPMC812 handle
    IOCTL_TP812_SYNCCLK_ENA, // control code
    NULL,                    // no input buffer
    0,
    NULL,                    // no receive return values
    0,
    &NumBytes,               // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Enable Master Clk failed"); // process error
}

...
```

#### See Also

Win32 documentation `DeviceIoControl()`, TPMC812 Hardware User Manual

### 3.1.3.9 IOCTL\_TP812\_SYNCCLK\_DISA

This control function disables the output of the master synchronization clock of the TPMC812 associated with the open device handle.

Because there are no parameters needed, the parameter pointer `lpInBuffer` and `lpOutBuffer` must be set to `NULL`, the size of the buffers must be set to 0.

#### Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG           NumBytes;

...

// Disable master synchronization clock
success = DeviceIoControl(
    hCurrent,                // TPMC812 handle
    IOCTL_TP812_SYNCCLK_DISA, // control code
    NULL,                    // no input buffer
    0,
    NULL,                    // no receive return values
    0,
    &NumBytes,               // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("disable Master Clk failed"); // process error
}

...
```

#### See Also

Win32 documentation `DeviceIoControl()`, TPMC812 Hardware User Manual

### 3.1.3.10 IOCTL\_TP812\_READ\_ENC

This control function attempts to read the actual value and state of the both encoder registers of the TPMC812 associated with the open device handle.

There is no input parameter needed, `lpInBuffer` must be set to `NULL` and the size argument `nInBufferSize` must be set to 0.

The return buffer (`TP812_OUT_READ_ENC_BUF`) must be pointed by `lpOutBuffer`. The argument `nOutBufferSize` specifies the size (size of `TP812_OUT_READ_ENC_BUF`) of the read buffer.

The `TP812_OUT_READ_ENC_BUF` structure has the following layout (see `TPMC812.h`):

```
typedef struct{
    UCHAR          encValue1;
    UCHAR          encValue2;
    UCHAR          flags;
} TP812_OUT_READ_ENC_BUF, *PTP812_OUT_READ_ENC_BUF;
```

#### *encValue1*

This value returns the actual value of encoder channel 1. (Value range: 0 ... 255)

#### *encValue2*

This value returns the actual value of encoder channel 2. (Value range: 0 ... 255)

#### *flags*

This value returns a bit mask, indication the states of encoder channel 1 and 2. Following flags are defined in `TPMC812.h`.

Value	Description
<code>TP812_FL_OVERRUN1</code>	Encoder 1 indicates an overrun (255 → 0)
<code>TP812_FL_UNDERRUN1</code>	Encoder 1 indicated an underrun (0 → 255)
<code>TP812_FL_OVERRUN2</code>	Encoder 2 indicates an overrun (255 → 0)
<code>TP812_FL_UNDERRUN2</code>	Encoder 2 indicated an underrun (0 → 255)

## Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN         success;
ULONG          NumBytes;
TP812_OUT_READ_ENC_BUF  oRdEncBuf;

...

// Read encoder values
success = DeviceIoControl(
    hCurrent,          // TPMC812 handle
    IOCTL_TP812_READ_MEM, // control code
    NULL,             // no input buffer
    0,
    &oRdEncBuf,       // address of return value
    sizeof(oRdEncBuf),
    &NumBytes,       // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Encoder Read failed"); // process error
}

...
```

## Error Codes

STATUS_INVALID_PARAMETER	This error is returned if the size of the write buffer is too small.
--------------------------	--

## See Also

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

### 3.1.3.11 IOCTL\_TP812\_CLEAR\_ENC

This control function attempts to clear the encoder counter(s) of the TPMC812 associated with the open device handle.

The needed parameters are passed in a buffer (*TP812\_IN\_CLEAR\_REG\_BUF*), pointed by *lpInBuffer*, to the driver. The argument *nInBufferSize* specifies the size (size of *TP812\_IN\_CLEAR\_REG\_BUF*) of the write buffer.

The *TP812\_IN\_CLEAR\_REG\_BUF* structure has the following layout (see TPMC812.h):

typedef struct

```
{  
    BOOLEAN          cEnc1;  
    BOOLEAN          cEnc2;  
} TP812_IN_CLEAR_ENC_BUF, *PTP812_IN_CLEAR_ENC_BUF;
```

*cEnc1*

This argument specifies if encoder 1 should be cleared or not. A *TRUE* value will clear the encoder, a *FALSE* value not.

*cEnc2*

This argument specifies if encoder 2 should be cleared or not. A *TRUE* value will clear the encoder, a *FALSE* value not.

## Example

```
#include <windows.h>
#include <winioctl.h>
#include "TPMC812.h"

HANDLE          hDevice;
BOOLEAN        success;
ULONG          NumBytes;
TP812_IN_CLEAR_ENC_BUF iClEncBuf;

...

// Clear encoder 2
iClEncBuf.clEnc1 = FALSE;
iClEncBuf.clEnc2 = TRUE;
success = DeviceIoControl(
    hCurrent,                // TPMC812 handle
    IOCTL_TP812_CLEAR_ENC,  // control code
    &iClEncBuf,              // buffer with control
                            // information for the driver
    sizeof(iClEncBuf),
    NULL,                    // no receive return values
    0,
    &NumBytes,              // number of bytes transferred
    NULL);

If (!success)
{
    ErrorHandler("Encoder clear failed"); // process error
}

...
```

## Error Codes

STATUS\_INVALID\_PARAMETER      This error is returned if the size of the write buffer is too small.

## See Also

Win32 documentation DeviceIoControl(), TPMC812 Hardware User Manual

---

## 4 Example Applications

### 4.1 Simple Example Application

This example should show how to use device driver calls. This example has been compiled with Microsoft Visual C++ 6.0.

This example allows to

- Read and write register values of the SERCON-Chip.
- Copy buffers to and from the dual ported memory.
- Activate and deactivate the synchronous master clock output signal
- Read and clear the encoder values

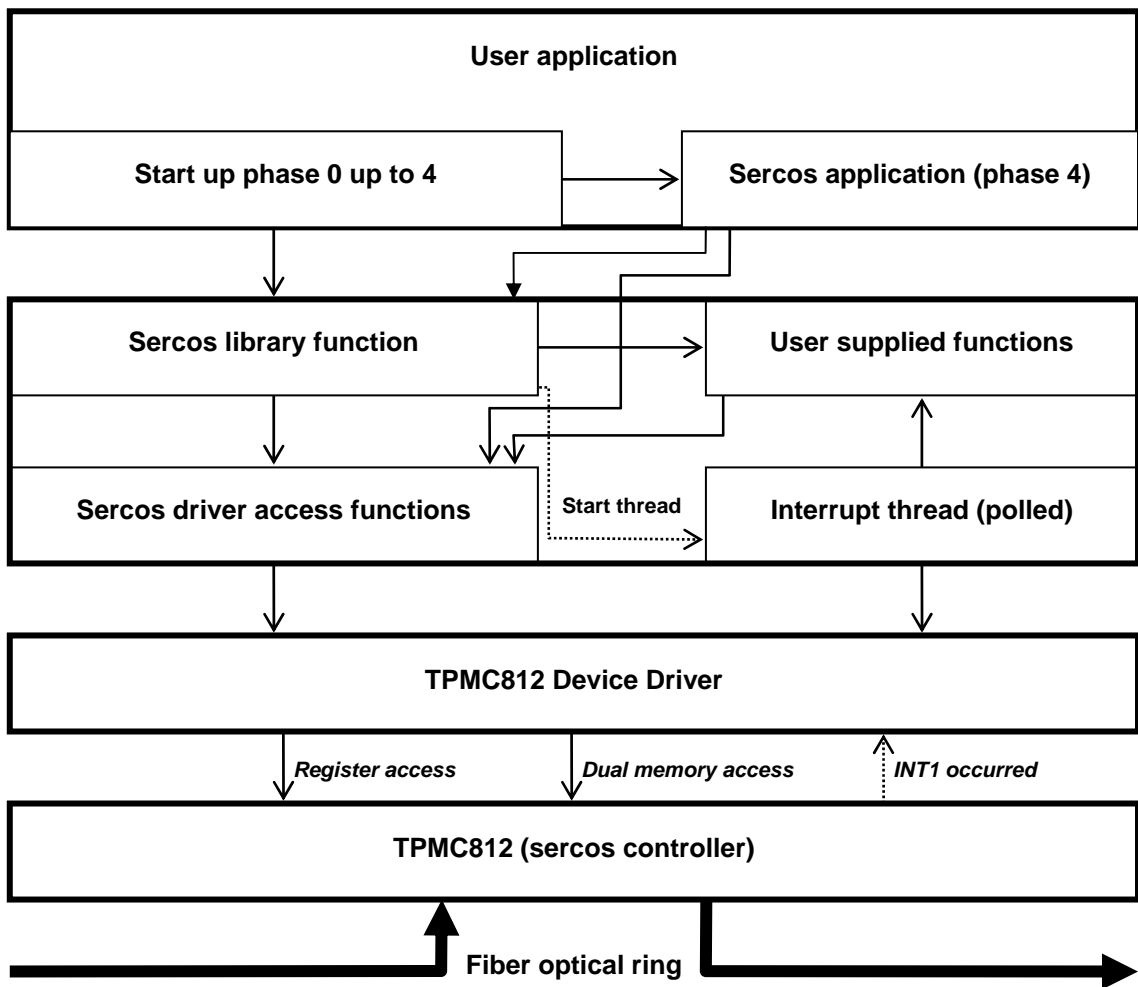
### 4.2 Sercos Master Example Application

This example shows how to start up a sercos master. This example has been compiled with Microsoft Visual C++ 6.0.

The example shows to start up a ring for a system with two simulated slave drives and will not work without changes in other systems.

The example is split into two main files. The first one (*SercosMaster.c*) is the application dependent part, it starts the ring through the five phases and then executes the application. The second file (*SercosLib.c*) contains the functions which depend on the sercos hardware. The functions needed by the application part will be described below. There are also some functions which must be supplied by the user. These functions will also be described below.

The following diagram shows how the different sercos layers are connected.



## 4.2.1 Sercos driver access functions

These functions allow a simpler access to the device driver. These functions are used by the library functions while starting up the ring. After reaching phase 4 the sercos application will also use these functions.

### 4.2.1.1 Write\_SC\_sercon\_reg

This function writes a value to a specified register.

```
ULONG Write_SC_sercon_reg
(
    PTP812_SC_CB_STRUCT pSercos_cb,
    USHORT registerOffset,
    USHORT usValue
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*registerOffset*

This parameter specifies the register that should be written. (See also TPMC812.h)

*usValue*

This parameter specifies the value that should be written.

#### Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

## Example

```
ULONG    RetError;  
USHORT   TmpWert;  
  
//  
//  read content of register 4  
//  
RetError = Write_SC_sercon_reg(pSercos_cb, TP812_REG04, TmpWert);  
if (RetError != 0) {  
    // Handle error  
}  
  
...
```

## See Also

See 3.1.3.1 IOCTL\_TP812\_WRITE\_REG.

### 4.2.1.2 Read\_SC\_sercon\_reg

This function performs a read access to a specified register.

```
ULONG Read_SC_sercon_reg  
(  
    PTP812_SC_CB_STRUCT pSercos_cb,  
    USHORT registerOffset,  
    PUSHORT pusValue  
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*registerOffset*

This parameter specifies the register that should be read. (See also TPMC812.h)

*pusValue*

This parameter points to a variable where the register value will be stored to.

#### Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

#### Example

```
ULONG    RetError;  
USHORT   TmpWert;  
  
//  
//  read content of register 4  
//  
RetError = Read_SC_sercon_reg(pSercos_cb, TP812_REG04, &TmpWert);  
if (RetError != 0) {  
    // Handle error  
}  
  
...
```

#### See Also

See 3.1.3.2 IOCTL\_TP812\_READ\_REG.

### 4.2.1.3 Write\_SC\_sercon\_mem

This function copies the content of a buffer to a specified dual ported memory address.

```
ULONG Write_SC_sercon_mem  
(  
    PTP812_SC_CB_STRUCT pSercos_cb,  
    USHORT memOffset,  
    USHORT length,  
    PUSHORT buffer  
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*memOffset*

This parameter specifies the offset of the start address in the dual ported memory the buffer should be copied to.

*length*

This parameter specifies the number of words to be copied.

*buffer*

This parameter points to the buffer where the data is stored that should be copied.

#### Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

## Example

```
ULONG    RetError;
USHORT   TmpBuffer[] = {0x1010,0x1111,0x1212,0x1313};

//
//  copy buffer to dual ported memory (address 0x100)
//  of the TPMC812
//
RetError = Write_SC_sercon_mem(
    pSercos_cb,
    0x100,
    4,
    &TmpBuffer[0]);
if (RetError != 0) {
    // Handle error
}

...
```

## See Also

See 3.1.3.3 IOCTL\_TP812\_WRITE\_MEM.

#### 4.2.1.4 Read\_SC\_sercon\_mem

This function copies the content of a specified dual ported memory address to a buffer.

```
ULONG Read_SC_sercon_mem  
(  
    PTP812_SC_CB_STRUCT pSercos_cb,  
    USHORT memOffset,  
    USHORT length,  
    PUSHORT buffer  
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*memOffset*

This parameter specifies the offset of the start address in the dual ported memory the buffer should be copied from.

*length*

This parameter specifies the number of words to be copied.

*buffer*

This parameter points to the buffer where the data shall be stored.

#### Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

## Example

```
ULONG    RetError;
USHORT   TmpBuffer[10];

//
//  copy buffer from dual ported memory
//  (size 0x10, address 0x100) of the TPMC812
//
RetError = Read_SC_sercon_mem(
    pSercos_cb,
    0x100,
    10,
    &TmpBuffer[0]);
if (RetError != 0) {
    // Handle error
}

...
```

## See Also

See 3.1.3.4 IOCTL\_TP812\_READ\_MEM.

## 4.2.2 User supplied functions

These functions have to be supported by the user. These functions are application dependent and can't be covered by default subroutines.

### 4.2.2.1 SercosMasterApplicationDataInfo

This function returns the length parameters of an existing data block of a specified ident number. For constant length the length is returned otherwise a variable block length is indicated.

```
INT SercosMasterApplicationDataInfo
(
    PTP812_SC_CB_STRUCT pSercos_cb,
    INT AtNumber,
    USHORT IdentNumber,
    INT ElementNumber
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*AtNumber*

This parameter specifies the drive.

*IdentNumber*

This parameter specifies the data block which length is needed.

*ElementNumber*

This parameter specifies the data element which length is needed.

#### Return Value

- 0 No data block exists for the specified *IdentNumber*
- < 0 The data length is variable. (The first word contains the actual length and the second word contains the maximal length of the following data.)
- > 0 The length is constant. (The return value contains the data length. Possible values are 1 and 2.)

### 4.2.2.2 TransferSercosMasterApplicationData

This function is used for transmitting service data between the Master and the drive. The data length is variable.

```
ULONG TransferSercosMasterApplicationData
(
    PTP812_SC_CB_STRUCT pSercos_cb,
    INT AtNumber,
    USHORT IdentNumber,
    INT ElementNumber,
    INT Direction,
    INT DataOffset,
    PUSHORT pBuffer,
    PUSHORT pDataLength,
    PINT pMoreData
)
```

#### Parameters

##### *pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

##### *AtNumber*

This parameter specifies the drive.

##### *IdentNumber*

This parameter specifies the data block type.

##### *ElementNumber*

This parameter specifies the data element.

##### *Direction*

This parameter specifies the direction of the transmission. Following values are allowed (defined in SERCOSLIB.h):

MASTER_TO_DRIVE	Transfer from master to drive
DRIVE_TO_MASTER	Transfer from drive to master

##### *DataOffset*

This parameter specifies the number of data words of the element that has already been exchanged.

##### *pBuffer*

This parameter points to the data buffer the data shall be copied to, if *Direction* is set to *MASTER\_TO\_DRIVE*. If *Direction* is set to *DRIVE\_TO\_MASTER* the parameter points to the buffer the data will be copied from.

#### *pDataLength*

This parameter points to a variable that specifies the length of the data buffer. If master to drive communication is selected, the value specifies the maximum length of the data buffer and the function has to change the value before returning to the real data length.

If a drive to master communication is selected this parameter specifies the real data length.

#### *pMoreData*

This parameter points to a variable that indicates if the actual block is the last block of the selected data element. If the value is zero, it means that the actual block is the last one of the data element. If the value is greater zero it indicates that some more blocks are needed for a complete transmission.

### **Return Value**

The function returns `NO_ERROR` on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

## 4.2.3 Sercos Library Functions

These functions are hardware and application independent. These functions will be used by the application and during start up.

### 4.2.3.1 InitSercosMaster

This function initializes the TPMC812 sercos module, sets the device data structure up, allocates and sets internal structures up and it starts the interrupt controlling thread. This function must be called before any other function for the ring is called.

```
ULONG InitSercosMaster  
(  
    PTP812_SC_CB_STRUCT pSercos_cb  
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

#### Example

```
ULONG MasterError;  
TP812_SC_CB_STRUCT serc_m_cb;  
  
MasterError = InitSercosMaster(&serc_m_cb);  
  
if (MasterError != NO_ERROR) {  
    // Handle Error  
}  
  
...
```

#### Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

### 4.2.3.2 StartSercosMasterPhase

This function tries to start up the communication of the ring to the selected phase. Only allowed phase switches will be accepted. (Allowed phase switches will be described later, see 4.2.6 Startup a Ring)

```
ULONG StartSercosMasterPhase
(
    PTP812_SC_CB_STRUCT pSercos_cb,
    USHORT PhaseNumber
)
```

#### Parameters

##### *pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

##### *PhaseNumber*

This parameter selects the phase number to start. Allowed values are 0 up to 4.

#### Example

```
ULONG MasterError;
PTP812_SC_CB_STRUCT pSercos_cb;

// Switch to phase 0
MasterError = StartSercosMasterPhase(pSercos_cb, 0);
if (MasterError != NO_ERROR)
{
    // Handle Error
}

...

// Switch to phase 1
MasterError = StartSercosMasterPhase(pSercos_cb, 1);
if (MasterError != NO_ERROR)
{
    // Handle Error
}

...
```

```
..

// Switch to phase 2
MasterError = StartSercosMasterPhase(pSercos_cb, 2);
if (MasterError != NO_ERROR)
{
    // Handle Error
}

...

// Switch to phase 3
MasterError = StartSercosMasterPhase(pSercos_cb, 3);
if (MasterError != NO_ERROR)
{
    // Handle Error
}

...

// Switch to phase 4
MasterError = StartSercosMasterPhase(pSercos_cb, 4);
if (MasterError != NO_ERROR)
{
    // Handle Error
}

...
```

### **Return Value**

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

### 4.2.3.3 GetSercosMasterPhaseStatus

This function returns the actual state of the last started phase change. This function could be called during or after the change of a phase.

```
ULONG GetSercosMasterPhaseStatus  
(  
    void  
)
```

#### Example

```
ULONG MasterError;  
  
MasterState = GetSercosMasterPhaseStatus();  
switch(MasterState)  
{  
    case READY:  
        ...  
        break;  
  
    case NOT_READY:  
        ...  
        break;  
  
    ...  
}  
  
...
```

#### Return Value

NOT_READY	phase change is processing, no error
READY	phase change completed successfully
All other values	An error occurred, an error code is returned. (See 4.2.7 Error Codes)

#### 4.2.3.4 StartSercosMasterDriveConnection

This function tries to start a direct communication between the master and the specified drive.

```
ULONG StartSercosMasterDriveConnection
(
    PTP812_SC_CB_STRUCT pSercos_cb,
    INT AtNumber
)
```

**This function is only usable in phase 1 and phase 2, because only these two phases allow a direct connection. This function is not interruptible, all interrupts have to be disabled. The function starts the connecting process, but it doesn't wait for completion. The completion must be detected with *GetSercosMasterConnectionStatus* function.**

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*AtNumber*

This parameter specifies the drive.

#### Example

```
ULONG MasterError;
PTP812_SC_CB_STRUCT pSercos_cb;

// Make connection with driver 1
MasterError = StartSercosMasterDriveConnection(
    pSercos_cb,
    1);
if (MasterError != NO_ERROR) {
    {
        // Handle Error
    }
}

...
```

#### Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

### 4.2.3.5 GetSercosMasterConnectionStatus

This function returns the actual state of the last started connection process.

```
ULONG GetSercosMasterConnectionStatus
(
    void
)
```

**This function is only valid if a connection has been started with *StartSercosMasterDriveConnection* function. This function should only be used in phase 1 and phase 2, because the *StartSercosMasterDriveConnection* function is only allowed in this two phases.**

### Example

```
ULONG MasterError;

MasterState = GetSercosMasterConnectionStatus();
switch(MasterState)
{
    case READY:
        ...
        break;

    case NOT_READY:
        ...
        break;

    ...
}

...
```

### Return Value

NOT_READY	connection process is active, no error
READY	connection process completed successfully, connection is valid
All other values	An error occurred, an error code is returned. (See 4.2.7 Error Codes)

### 4.2.3.6 StartSercosMasterServiceTransfer

This function starts a connection between the master and the specified drive.

```
ULONG StartSercosMasterServiceTransfer
(
    PTP812_SC_CB_STRUCT pSercos_cb,
    INT AtNumber,
    USHORT IdentNumber,
    BYTE ElementMask,
    BYTE Direction
)
```

**This function should not be used before phase 2 is reached. A started service transfer can be aborted with the function *AbortSercosMasterServiceTransfer* while the transfer status is active. The actual state of the transfer can be read out with *GetSercosMasterServiceStatus* function.**

#### Parameters

##### *pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

##### *AtNumber*

This parameter specifies the drive.

##### *IdentNumber*

This parameter selects the data to send.

##### *ElementMask*

This parameter specifies the data segments that shall be transmitted.

##### *Direction*

This parameter specifies the direction of the transmission. Following values are allowed (defined in SERCOSLIB.h):

MASTER_TO_DRIVE	Transfer from master to drive
DRIVE_TO_MASTER	Transfer from drive to master

## Example

```
ULONG MasterError;
PTP812_SC_CB_STRUCT pSercos_cb;

// Start transfer from drive 1 to master
// Ident Number 95, Element 7
MasterError = StartSercosMasterServiceTransfer(
    pSercos_cb,
    1,
    95,
    0x80,
    DRIVE_TO_MASTER);
if (MasterError == NO_ERROR)
{
    // Handle Error
}

...
```

## Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

#### 4.2.3.7 AbortSercosMasterServiceTransfer

This function aborts a connection between the master and the specified drive.

```
ULONG AbortSercosMasterServiceTransfer  
(  
    INT AtNumber  
)
```

**This function quits a service transfer which was previously started with *StartSercosMasterServiceTransfer* function.**

#### Parameters

*AtNumber*

This parameter specifies the drive.

#### Example

```
ULONG MasterError;  
  
// Abort transfer with drive 1  
MasterError = AbortSercosMasterServiceTransfer (1);  
if (MasterError == NO_ERROR)  
{  
    // Handle Error  
}  
  
...
```

#### Return Value

The function returns always NO\_ERROR.

#### 4.2.3.8 GetSercosMasterServiceStatus

This function returns the actual state of the last started service data transfer.

```
ULONG GetSercosMasterServiceStatus  
(  
    PTP812_SC_CB_STRUCT pSercos_cb,  
    INT AtNumber,  
    PUSHORT pSCErrorState,  
    PUSHORT pSCReceipt,  
    PINT pSCChange  
)
```

**This function is only valid if a service transfer is started with *StartSercosMasterServiceTransfer* function for this drive.**

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

*AtNumber*

This parameter specifies the drive.

*pSCErrorState*

The parameter points to a variable, where the drive service information or the error code is stored. Possible error codes are *ERROR\_NODATA* and *ERROR\_SCHSTIMEOUT*. (See 4.2.7 Error Codes)

*pSCReceipt*

This parameter points to variable, where the data state is stored.

*pSCChange*

This parameter points to variable that holds the command change bit.

## Example

```
ULONG MasterError;
PTP812_SC_CB_STRUCT pSercos_cb;
USHORT ErrorState,
USHORT CmdReceipt,
INT CmdChange)

// Get status from service with drive 1
MasterError = GetSercosMasterServiceStatus(
    pSercos_cb,
    1,
    &TmpSCErrorState,
    &TmpSCReceipt,
    &TmpSCChange);

switch (MasterError) {
case READY:
    // Service completed
    ...

case NOT_READY:
    // Service is active
    ...

case ...:
    ...
}

...
```

## Return Value

NOT_READY	connection process is active, no error
READY	connection process completed successfully, connection is valid
NOT_READY_BUT_BUSYTIMEOUT	data transfer active but the time limit is passed.
All other values	An error occurred, an error code is returned. (See 4.2.7 Error Codes)

### 4.2.3.9 PrepareSercosMasterForPhase3

This function calculates the communication parameters for phase 3 and phase 4. Phase 3 and phase 4 communication cycles are split into time slices. This function calculates all time slices.

```
ULONG PrepareSercosMasterForPhase3
(
    PTP812_SC_CB_STRUCT pSercos_cb
)
```

#### Parameters

*pSercos\_cb*

This parameter is a pointer to a structure (TP812\_SC\_CB\_STRUCT) that selects the sercos controller. (The structure TP812\_SC\_CB\_STRUCT will be described later, see 4.2.4.1 TP812\_SC\_CB\_STRUCT)

#### Other Information

This function needs the following values filled in the data structure *pUserRingInfo* which is part of *TP812\_SC\_CB\_STRUCT*:

##### Filled by user:

```
ptr = pSercos_cb->pUserRingInfo;
ptr->AtCount = ...;
ptr->Datarate = ...;
ptr->pATUserTime[0..last_drive - 1]->Tscyc = ...;
ptr->pATUserTime[0..last_drive - 1]->CyclicDataCount = ...;
ptr->pMdtUserTime[0..last_drive - 1]->CyclicDataCount = ...;
```

##### Read over service channel:

```
ptr->pATUserTime[0..last_drive - 1]->Tlmin = ...;
ptr->pATUserTime[0..last_drive - 1]->T4min = ...;
ptr->pATUserTime[0..last_drive - 1]->Tmtsg = ...;
ptr->pATUserTime[0..last_drive - 1]->Tmtsy = ...;
ptr->pATUserTime[0..last_drive - 1]->Tatmt = ...;
ptr->pATUserTime[0..last_drive - 1]->TSlavekennung = ...;
ptr->pATUserTime[0..last_drive - 1]->Tatat = ...;
(if more then one drive at slave)
```

After calculating the values, they have to be written to the drives using the service channel.

Calculated values:

```
ptr->pATUserTime[0..last_drive - 1]->T1
ptr->pATUserTime[0..last_drive - 1]->T2
ptr->pATUserTime[0..last_drive - 1]->T3
ptr->pATUserTime[0..last_drive - 1]->T4
ptr->pATUserTime[0..last_drive - 1]->LengthMasterData
ptr->pATUserTime[0..last_drive - 1]->OffsetMasterData
```

## Example

```
ULONG MasterError;
PTP812_SC_CB_STRUCT pSercos_cb;

// Initialize structure
...

MasterError = PrepareSercosMasterForPhase3(pSercos_cb);
if (MasterError == NO_ERROR)
{
    // Handle Error
}

...
```

## Return Value

The function returns NO\_ERROR on success, otherwise it returns an error code. (See 4.2.7 Error Codes)

## 4.2.4 Data Structures

### 4.2.4.1 TP812\_SC\_CB\_STRUCT

This structure contains all information needed for the ring. Some information are just needed for the application start up or needed for special phases, these parts may be changed for other application and there are some parameters that are needed for sercos library functions and that must be filled by the application. The information that is needed by the library will be described below.

```
typedef struct
{
    // Driver connection
    HANDLE ringHandle;

    // Master Error State
    ULONG MasterError;

    // Master Phase State
    BYTE PhaseNumber;
    ENUM_PhaseState PhaseState0;
    ENUM_PhaseState PhaseState1;
    ENUM_PhaseState PhaseState2;
    ENUM_PhaseState PhaseState3;
    ENUM_PhaseState PhaseState4;

    //
    PUSERRINGINFO pUserRingInfo;

    // Phase dependent, static variables
    TP812_SC_PH1 ph1;
    TP812_SC_PH2 ph2;
    TP812_SC_PH3 ph3;
    TP812_SC_PH4 ph4;
} TP812_SC_CB_STRUCT, *PTP812_SC_CB_STRUCT;
```

#### *ringHandle*

This value specifies the device that shall be used and must be filled by the application. The handle selects a special device, which makes it possible to use more than one ring. How to get the handle can be seen in the sercos master example application.

*MasterError*

This value shows the last error occurred. (For error codes see 4.2.7 Error Codes)

*PhaseNumber*

This value shows the actual phase.

*PhaseState0*

*PhaseState1*

*PhaseState2*

*PhaseState3*

*PhaseState4*

These entries show the actual state of the different phases.

*pUserRingInfo*

This entry points to the *USERRINGINFO* structure. (see 4.2.4.2 USERRINGINFO)

*ph1*

*ph2*

*ph3*

*ph4*

These entries are application dependent information and are not needed by the library functions.

#### 4.2.4.2 USERRINGINFO

The data exchange between user program and library functions is handled with the data structure *pUserRingInfo*, which is part of *TP812\_SC\_CB\_STRUCT*. There are two kinds of values in the structure *USERRINGINFO*, first the values which are read only, they are marked with *RO*, and the registers which could be written, they are marked with *WR*.

```
typedef struct
{
    int AtCount;                // W
    int Datarate;              // W
    PATUSERDATA pAtUserData[MAXAT];
    PATUSERTIME pAtUserTime[MAXAT];
    PMDTUSERDATA pMdtUserData[MAXAT];
    USHORT MdtHeaderOffset;    // R
    USHORT EndHeaderOffset;    // R
} USERRINGINFO, *PUSERRINGINFO;
```

##### *AtCount*

This value specifies the number of drives which are part of the ring.

##### *Datarate*

This value specifies the data ring that will be used in the ring.

##### *pAtUserData*

##### *pAtUserTime*

##### *pMdtUserData*

These pointers point to data structures that will be described below in this chapter.

##### *MdtHeaderOffset*

This value specifies the offset of the MDT header in the dual ported memory.

##### *EndHeaderOffset*

This value specifies the offset of the end header in the dual ported memory.

#### 4.2.4.3 ATUSERDATA

```
typedef struct
{
    int AtAddress;                // W - Address of drive
    int CyclicDataCount;         // W - Count of cyclic data (USHORT)
                                //   without info and control
    USHORT HeaderOffset;        // R - Start of At header
    USHORT DataContainerOffset; // R - Start of At data container
    int DataContainerLength;     // R - Double buffer and control
                                //   (USHORT)
} ATUSERDATA, *PATUSERDATA;
```

##### *AtAddress*

This value specifies the address of the drive.

##### *CyclicDataCount*

This value specifies the number of words for cyclic data. (Without info and control)

##### *HeaderOffset*

This value holds the offset of the At header in the dual ported memory.

##### *DataContainerOffset*

This value holds the offset of the At data container in the dual ported memory.

##### *DataContainerLength*

This value holds the length in words of the At header in the dual ported memory.

#### 4.2.4.4 ATUSERTIME

```
typedef struct
{
    USHORT T1min;           // W
    USHORT Tatat;          // W
    USHORT Tatmt;          // W
    USHORT Tmtsy;          // W
    USHORT Tmtsg;          // W
    USHORT T4min;          // W
    USHORT Tscyc;          // W
    USHORT Tncyc;          // W
    USHORT Slavedetection; // W
    USHORT T1;             // R
    USHORT T2;             // R
    USHORT T3;             // R
    USHORT T4;             // R
    USHORT LengthMasterData; // R - Value in BYTE
    USHORT OffsetMasterData; // R - Value in BYTE
} ATUSERTIME, *PATUSERTIME;
```

*T1min*

*Tatat*

*Tatmt*

*Tmtsy*

*Tmtsg*

*T4min*

*Tscyc*

*Tncyc*

*Slavedetection*

*T1*

*T2*

*T3*

*T4*

The values are used, calculated and set when calculating the ring timing.

*LengthMasterData*

This value specifies the data length in bytes in the MDT.

*OffsetMasterData*

This value specifies the offset in bytes of the data set in the MDT.

#### 4.2.4.5 MDTUSERDATA

```
typedef struct
{
    USHORT DataContainerOffset;           // R
    int DataContainerLength;             // R - Double buffer and control
                                        // (USHORT)
    int CyclicDataCount;                 // W - Count of cyclic data (USHORT)
                                        // without info and control
} MDTUSERDATA, *PMDTUSERDATA;
```

##### *DataContainerOffset*

This value specifies the offset of the data container in the dual ported memory.

##### *DataContainerLength*

This value specifies the length in words of the data container (double buffer).

##### *CyclicDataCount*

This value specifies the length of cyclic data. (Without info and control)

## 4.2.5 Definitions (Constant Values)

### 4.2.5.1 Baudrate

Value	Name	Description
1	BAUD_2M	Select 2 Mbit communication rate
0	BAUD_4M	Select 4 Mbit communication rate

### 4.2.5.2 State Message

Value	Name	Description
0x0000	READY	The last process completed and no error was found. (Example: Phase change: Phase change completed successfully).
0x7FFF	NOT_READY	The last process is still busy (no error).
0x70FF	NOT_READY_BUT_BUSYTIMEOUT	The last process has not completed, but an error was found.
0x1111	READY_FOR_SCDATA	The process indicates that it is ready to get service data.

### 4.2.5.3 Data Direction

Value	Name	Description
0	MASTER_TO_DRIVE	Transmission from master to drives
1	DRIVE_TO_MASTER	Receive from drives to master

### 4.2.5.4 Phases

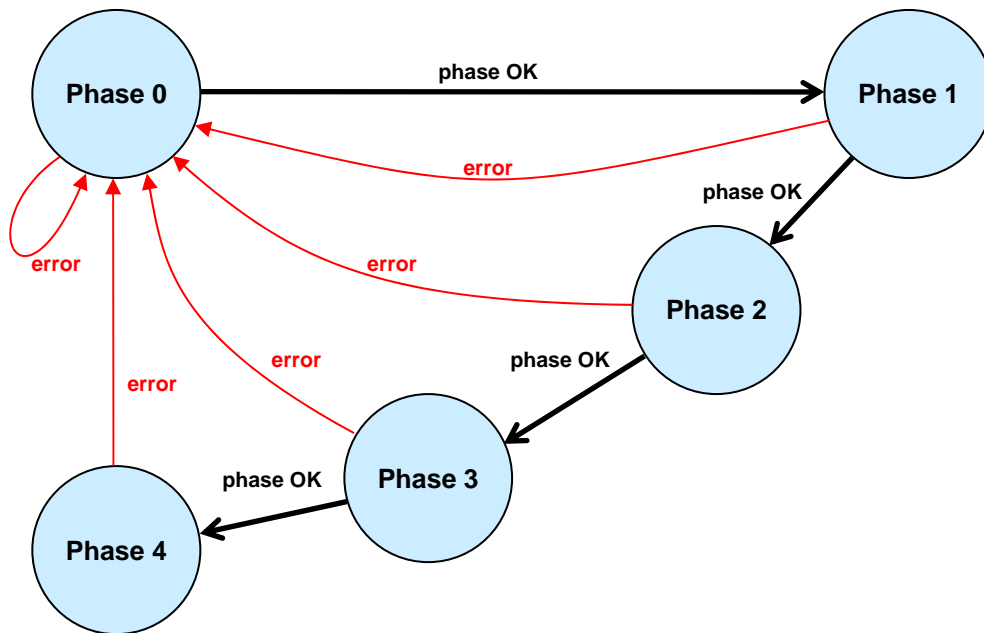
Value	Name	Description
0	PHASE0	Phase 0
1	PHASE1	Phase 1
2	PHASE2	Phase 2
3	PHASE3	Phase 3
4	PHASE4	Phase 4
5	DRIVER_INIT	Initialization phase

## 4.2.6 Startup a Ring

This chapter gives a short description of starting the communication of the ring. After reaching phase 4 the communication over the ring is full available.

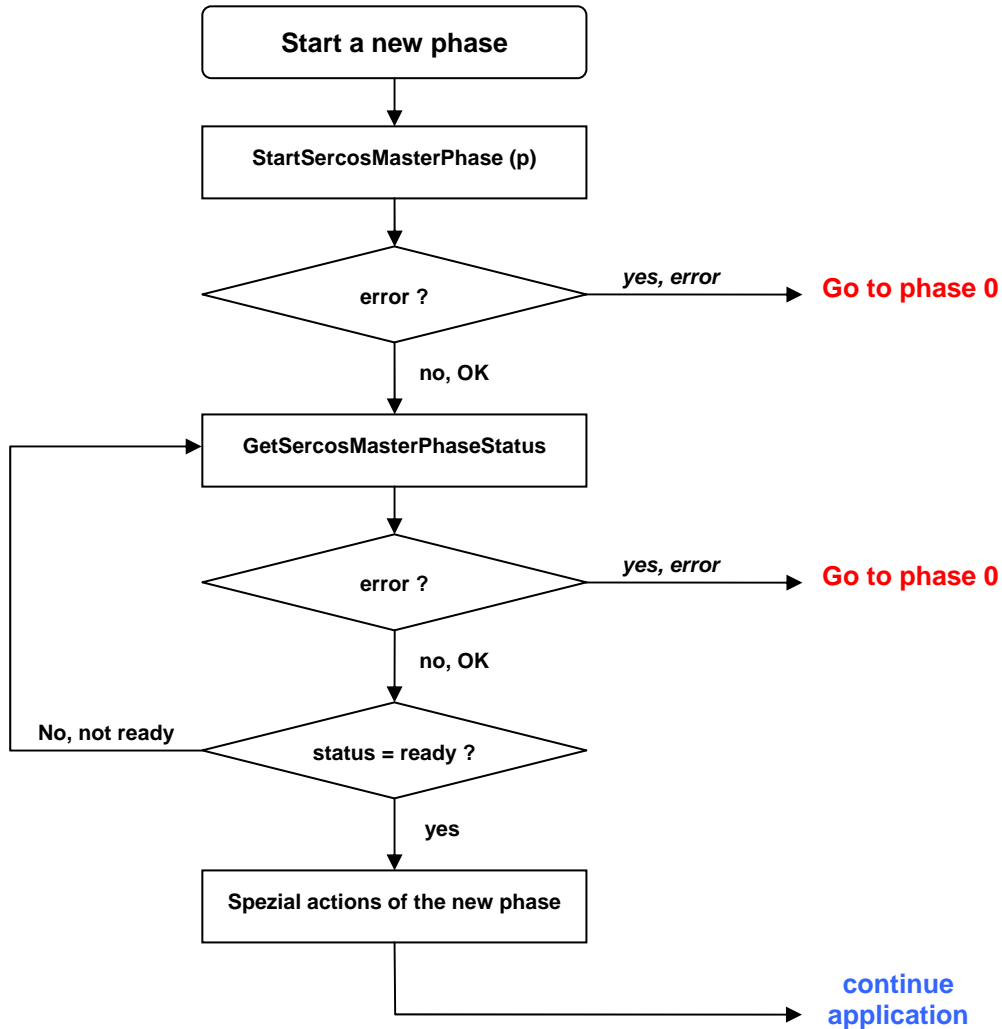
### 4.2.6.1 Error Handling

The startup have to start with phase 0 and go over the phases 1, 2 and 3 to phase 4, if any phase detects an error the startup must start again with phase 0.



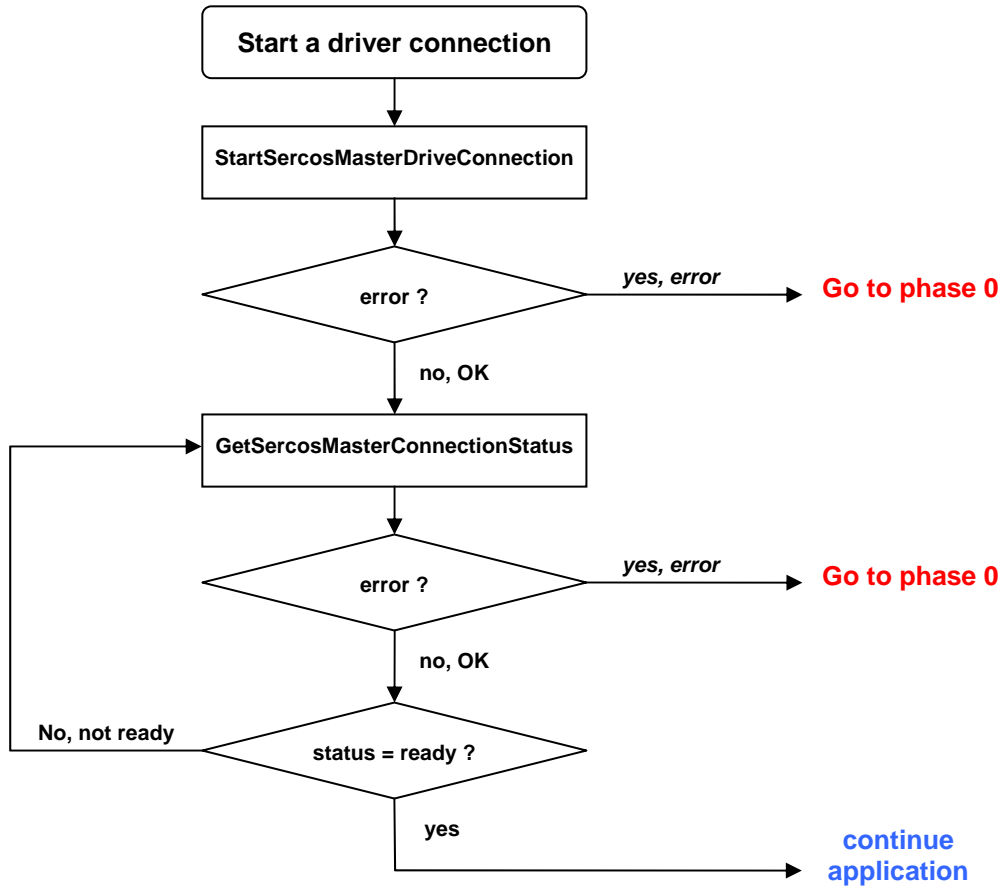
### 4.2.6.2 Start a new Phase

Starting a new phase is divided into three main parts, first the phase switch, second is waiting on phase change completion and third the actions which have to take place in the phase. These actions are defined by the sercos specification. (Example: in phase 1 the master has to check if all drives are answering over the bus.)



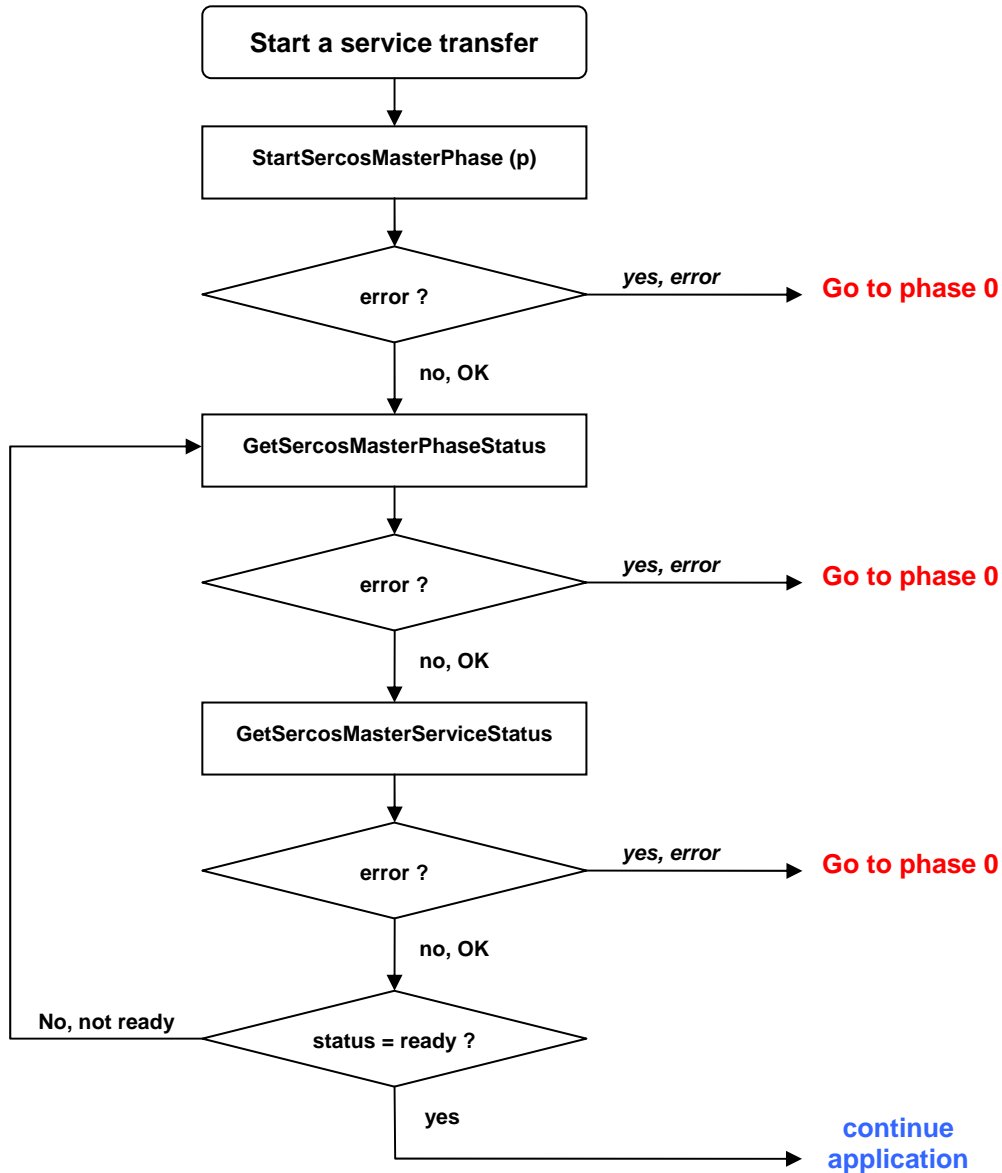
### 4.2.6.3 Start a Drive Connection

Starting a drive connection is only allowed in phase 1 and phase 2. In phase 1 it is used to check if all drives are answering. In phase 2 it is used to exchange the communication parameters. After starting a drive connection, the application has to wait until the connection is complete.



#### 4.2.6.4 Start a Service transfer

The service transfers can be started when phase 2 or a higher is running. Service is used to send parameters or other data to the selected drive. After starting a service transfer the application must check the connection and the communication, if there is an error or if the communication is complete.



## 4.2.7 Error Codes

Value	Error	Description
0x0000	NO_ERROR	No error
0x0001	ERROR_NOMEMORY	There is not enough memory for internal data structures.
0x0002	ERROR_SETINTVEC	The interrupt vector is missing or not OK.
0x0003	ERROR_HSTIMEOUT	Host timed out, no AT has been received during ten MDTs.
0x0004	ERROR_ATMISS	Two or more ATs are missing.
0x0005	ERROR_WRONGPHASE	The selected phase is not reachable from the actual state.
0x0006	ERROR_WRONGADDRESS	The slave address is not allowed.
0x0007	ERROR_WRONGATCOUNT	The selected ring number is not placed in the allowed area.
0x0009	ERROR_WRONGATNUMBER	Illegal or not allowed slave number.
0x0010	ERROR_DPRAMOVERFLOW	The memory area for cyclic data is smaller than the needed.
0x0011	ERROR_SCNOTINIT	The memory area of data channels is too small.
0x0012	ERROR_SCTRANSNOTREADY	The previous started data transmission is not ready.
0x0013	ERROR_SCTRANSNODATA	No data available for data channel.
0x0014	ERROR_CALCULATE_T2	The telegrams do not fit to the telegram timing or the cycle time is too short.
0x0015	ERROR_CALCULATE_T3	The telegrams do not fit to the telegram timing or the cycle time is too short.
0x0016	ERROR_CALCULATE_T4	The telegrams do not fit to the telegram timing or the cycle time is too short.
0x0017	ERROR_CALCULATE_TEND	The telegrams do not fit to the telegram timing, or the cycle time is too short.
0x0019	ERROR_SCNODATA	No data available
0x0020	ERROR_SCHSTIMEOUT	Service channel has timed out
0x0100	ERROR_HWACCESS	Error while hardware access detected.
0xFFFF	ERROR_DEFAULT	All other errors