```
///////////////////////////////////////////////////////////////////////////////
// TcAsyncBufferWritingModule.cpp
#include "TcPch.h"
#pragma hdrstop

#include "TcAsyncBufferWritingModule.h"
#include "TcTimeConversion.h"              //external dependency from Beckhoff to get and convert system time

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
DEFINE_THIS_FILE()

///////////////////////////////////////////////////////////////////////////////
// CTcAsyncBufferWritingModule
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Collection of interfaces implemented by module CTcAsyncBufferWritingModule
BEGIN_INTERFACE_MAP(CTcAsyncBufferWritingModule)
    INTERFACE_ENTRY_ITCOMOBJECT()
    INTERFACE_ENTRY(IID_ITcCyclic, ITcCyclic)
    INTERFACE_ENTRY(IID_ITcADI, ITcADI)
///<AutoGeneratedContent id="InterfaceMap">
    INTERFACE_ENTRY(IID_ITcCyclic, ITcCyclic)
///</AutoGeneratedContent>
END_INTERFACE_MAP()

IMPLEMENT_ITCOMOBJECT(CTcAsyncBufferWritingModule)
IMPLEMENT_ITCOMOBJECT_SETSTATE_LOCKOP2(CTcAsyncBufferWritingModule)
IMPLEMENT_ITCADI(CTcAsyncBufferWritingModule)

///////////////////////////////////////////////////////////////////////////////
// Set parameters of CTcAsyncBufferWritingModule
BEGIN_SETOBJPARA_MAP(CTcAsyncBufferWritingModule)
    SETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="SetObjectParameterMap">
    SETOBJPARA_VALUE(PID_TcAsyncBufferWritingModuleParameter, m_Parameter)
    SETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
END_SETOBJPARA_MAP()

///////////////////////////////////////////////////////////////////////////////
// Get parameters of CTcAsyncBufferWritingModule
BEGIN_GETOBJPARA_MAP(CTcAsyncBufferWritingModule)
    GETOBJPARA_DATAAREA_MAP()
///<AutoGeneratedContent id="GetObjectParameterMap">
    GETOBJPARA_VALUE(PID_TcAsyncBufferWritingModuleParameter, m_Parameter)
    GETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
///</AutoGeneratedContent>
END_GETOBJPARA_MAP()

///////////////////////////////////////////////////////////////////////////////
// Get data area members of CTcAsyncBufferWritingModule
BEGIN_OBJDATAAREA_MAP(CTcAsyncBufferWritingModule)
///<AutoGeneratedContent id="ObjectDataAreaMap">
    OBJDATAAREA_VALUE(ADI_TcAsyncBufferWritingModuleInputs, m_Inputs)
    OBJDATAAREA_VALUE(ADI_TcAsyncBufferWritingModuleOutputs, m_Outputs)
///</AutoGeneratedContent>
END_OBJDATAAREA_MAP()


///////////////////////////////////////////////////////////////////////////////
CTcAsyncBufferWritingModule::CTcAsyncBufferWritingModule()
{
```

```
    memset(&m_Parameter, 0, sizeof(m_Parameter));
    memset(&m_Inputs, 0, sizeof(m_Inputs));
    memset(&m_Outputs, 0, sizeof(m_Outputs));

    memset(&m_Buffer1, 0, sizeof(m_Buffer1));
    memset(&m_Buffer2, 0, sizeof(m_Buffer2));

    m_pBufferFill = m_Buffer1;
    m_nBufferFillIndex = 0;
    m_nBufferFillIndex2 = 0;
    m_pBufferWrite = NULL;
    titleflag = TRUE;
    m_EventBufferFillFlag = FALSE;
    Conti1_Event0 = TRUE;
    EventLastCycle = FALSE;
    init = TRUE;
}

//////////////////////////////////////////////////////////////////////
CTcAsyncBufferWritingModule::~CTcAsyncBufferWritingModule()
{
}


//////////////////////////////////////////////////////////////////////
// State Transitions
//////////////////////////////////////////////////////////////////////
IMPLEMENT_ITCOMOBJECT_SETOBJSTATE_IP_PI(CTcAsyncBufferWritingModule)

//////////////////////////////////////////////////////////////////////
// State transition from PREOP to SAFEOP
//
// Initialize input parameters
// Allocate memory
HRESULT CTcAsyncBufferWritingModule::SetObjStatePS(PTComInitDataHdr pInitData)
{
    HRESULT hr = S_OK;

    IMPLEMENT_ITCOMOBJECT_EVALUATE_INITDATA(pInitData);
    hr = m_spSrv->TcCreateInstance(CID_TcFileAccessAsync, m_spFileAccessAsync.GetIID(), (PPVOID)(&
    m_spFileAccessAsync));

    if (SUCCEEDED(hr))
    {
        ITComObjectPtr spFileAccessObj = m_spFileAccessAsync;

        OTCID oid = m_spCyclicCaller.GetOID();
        hr = FAILED(hr) ? hr : spFileAccessObj->TcSetObjPara(PID_TcFileAccessAsyncAdsProvider, sizeof(oid),
     &oid);
        if (m_Parameter.SegmentSize == 0)
        {
            hr = ADS_E_INVALIDPARM;
        }
        hr = FAILED(hr) ? hr : spFileAccessObj->TcSetObjPara(PID_TcFileAccessAsyncSegmentSize, sizeof
    (m_Parameter.SegmentSize), &m_Parameter.SegmentSize);
        hr = FAILED(hr) ? hr : spFileAccessObj->TcSetObjPara(PID_TcFileAccessAsyncTimeoutMs, sizeof
    (m_Parameter.Timeout), &m_Parameter.Timeout);
        hr = FAILED(hr) ? hr : spFileAccessObj->TcSetObjState(TCOM_STATE_SAFEOP, m_spSrv, NULL);
    }

    TRACE(FTEXT("hr=0x%08x"), hr);
    return hr;
}

//////////////////////////////////////////////////////////////////////
// State transition from SAFEOP to OP
```

```
//
// Register with other TwinCAT objects
HRESULT CTcAsyncBufferWritingModule::SetObjStateSO()
{
    HRESULT hr = S_OK;

    hr = AddModuleToCaller();

    // TODO: Add any additional initialization
    if (SUCCEEDED(hr))
    {
        ITComObjectPtr spFileAccessObj = m_spFileAccessAsync;
        hr = spFileAccessObj->TcSetObjState(TCOM_STATE_OP, m_spSrv, NULL);
    }

    // Cleanup if transition failed at some stage
    if ( FAILED(hr) )
    {
        RemoveModuleFromCaller();
    }
    return hr;
}

////////////////////////////////////////////////////////////////////////////
// State transition from OP to SAFEOP
HRESULT CTcAsyncBufferWritingModule::SetObjStateOS()
{
    HRESULT hr = S_OK;

    RemoveModuleFromCaller();

    return hr;
}

////////////////////////////////////////////////////////////////////////////
// State transition from SAFEOP to PREOP
HRESULT CTcAsyncBufferWritingModule::SetObjStateSP()
{
    HRESULT hr = S_OK;
    return hr;
}

#define TIMESTAMPSTR "%04d-%02d-%02d-%02d-%02d-%02d-%03d"  //type of timestamp "YYYY-mm-dd-HH-mm-ss-SSS"
#define TIMESTAMPPARM(st) st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond, st.wMilliseconds ↙
        //structure of timestamp

////////////////////////////////////////////////////////
// ITcCyclic
///<AutoGeneratedContent id="ImplementationOf_ITcCyclic">
HRESULT CTcAsyncBufferWritingModule::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context) ↙
    //where the main programm starts in loop
{
    HRESULT hr = S_OK;

    //make title file

    if (titleflag) {
        hr = m_fsmFileTitleWriter.Init
        (
            m_spSrv,
            m_spFileAccessAsync,
            "%TC_BOOTPRJPATH%TTITLETEST.txt",       //name and path to the to be saved file, %            ↙
    TC_BOOTPRJPATH% ist the settled BOOT file of TwinCAT3: C:\TwinCAT\3.1\Boot
            reinterpret_cast<PVOID>(Title),
            A.length() * sizeof(char)
        );
```

```
    }
    if (SUCCEEDED(hr))
    {
        titleflag = NULL;   // titleflag is only recorded once
    }
    if (m_fsmFileTitleWriter.IsActive())
    {
        hr = m_fsmFileTitleWriter.Eval();
    }


    //to get the system time: UTC at "begin of task" in 100 ns intervals since 1.1.1601
    hr = FAILED(hr) ? hr : ipTask->GetCurrentSysTime(&systime);
    SYSTEMTIME stSysTime;
    TcFileTimeToSystemTime(systime, &stSysTime);        //convert timestamp to timestamp structure
    sprintf(m_szTaskSysTime, TIMESTAMPSTR, TIMESTAMPPARM(stSysTime));       //convert timestamp structure
    to TIMESTAMPSTR

    //get nameand path of files: %TC_BOOTPRJPATH%timestamps+ATEST.txt
    sprintf(bu, "%s%s", m_szTaskSysTime, "ATEST.txt");
    sprintf(buf, "%s%s", "%TC_BOOTPRJPATH%", bu);
    name = buf;

    if (Conti1_Event0)
    {
        //if in continue recording mode
        if (FillBuffer(m_pBufferFill, ASYNCWRITE_ContiBUFFERSIZE, m_nBufferFillIndex))      //function
    FillBuffer is called
        {

            // this indicates that the buffer is full, switch to other buffer
            if (m_nBufferFillIndex == 0)
            {
                if (m_pBufferFill == m_Buffer1)
                {
                    m_pBufferFill = m_Buffer2;      //m_pBufferFill points to the to be filled Buffer
                    m_pBufferWrite = m_Buffer1;     //m_pBufferWrite points to the to be written Buffer
                }
                else
                {
                    m_pBufferFill = m_Buffer1;
                    m_pBufferWrite = m_Buffer2;     //if Buffer is full, m_pBufferWrite points to the to be
     written Buffer
                }
            }
        }

        // initialize the writer function
        if (m_pBufferWrite)
        {
            hr =
                m_fsmFileWriter.Init
                (
                    m_spSrv,
                    m_spFileAccessAsync,
                    name,
                    reinterpret_cast<PVOID>(m_pBufferWrite),
                    ASYNCWRITE_ContiBUFFERSIZE * sizeof(st_Buffer)
                );
        }
        if (SUCCEEDED(hr))
        {
            m_pBufferWrite = NULL;
        }
    }
    else
```

```
    {
        //if in eventbased recording mode
        if (init)
        {
            m_pBufferFill = m_Buffer3;      //the first Buffer to be filled is Buffer3
            init = FALSE;
        }
        if (FillBuffer(m_pBufferFill, ASYNCWRITE_EventBUFFERSIZE, m_nBufferFillIndex2))
        {
            if (m_EventBufferFillFlag)
                // start to record the 20 records after event
            {
                ++Flag;
                if (Flag == ASYNCWRITE_EventBUFFERSIZE)
                {
                    //if the 20 records are all recorded, change the buffer
                    if (m_pBufferFill == m_Buffer3)
                    {
                        m_pBufferWrite = m_Buffer3;
                        m_pBufferFill = m_Buffer4;
                    }
                    else
                    {
                        m_pBufferWrite = m_Buffer4;
                        m_pBufferFill = m_Buffer3;
                    }
                    m_EventBufferFillFlag = FALSE;
                    Flag = ASYNCWRITE_EventBUFFERSIZE+1;
                }
            }
            else
            {
                if (m_Inputs.Event)
                    //to get the up trigger of event signal and save the trigger in EventFlanke
                {
                    if (!EventLastCycle)
                    {
                        EventFlanke = TRUE;
                    }
                }
                EventLastCycle = m_Inputs.Event;

                if (EventFlanke)
                {
                    //if the event happens (up trigger detected), values in the buffer now are the 20 ↙
    records before the event
                    EventFlanke = FALSE;
                    m_nBufferFillIndex2 = 0;
                    //change buffer to be filled and written
                    if (m_pBufferFill == m_Buffer3)
                    {
                        m_pBufferWrite = m_Buffer3;
                        m_pBufferFill = m_Buffer4;
                    }
                    else
                    {
                        m_pBufferWrite = m_Buffer4;
                        m_pBufferFill = m_Buffer3;
                    }
                    m_EventBufferFillFlag = TRUE;       //have to record the next 20 records after event
                    Flag = 0;
                }
            }
        }
        // initialize the writer function
        if (m_pBufferWrite)
```

```
        {
            hr =
                m_fsmFileWriter.Init
                (
                    m_spSrv,
                    m_spFileAccessAsync,
                    name,
                    reinterpret_cast<PVOID>(m_pBufferWrite),
                    ASYNCWRITE_EventBUFFERSIZE * sizeof(st_Buffer)
                );
        }
        if (SUCCEEDED(hr))
        {
            m_pBufferWrite = NULL;
        }

    }

    //write records in the buffer to binary file
    if (m_fsmFileWriter.IsActive())
    {
        m_Outputs.WriteActive = TRUE;
        hr = m_fsmFileWriter.Eval();
        if (SUCCEEDED(hr))
        {
            m_Outputs.BytesWritten = m_fsmFileWriter.GetBytesWrittenTotal();
        }
    }
    else
    {
        m_Outputs.WriteActive = FALSE;
    }
    return hr;

}
///</AutoGeneratedContent>

//use the values from Inputs (connected to Outputs of Simulink Modul) to fill the buffer
BOOL CTcAsyncBufferWritingModule::FillBuffer
(
    st_Buffer* pBuffer,
    UINT nBuffer,
    UINT& nBufferIndex
)
{

    if (nBufferIndex<nBuffer)
    {
        pBuffer[nBufferIndex].Timestamp = systime;
        pBuffer[nBufferIndex].a = m_Inputs.a;
        pBuffer[nBufferIndex].b = m_Inputs.b;
        pBuffer[nBufferIndex].b2 = m_Inputs.b2;
        pBuffer[nBufferIndex].roomtemp1 = m_Inputs.Troom1;
        pBuffer[nBufferIndex].roomtemp2 = m_Inputs.Troom2;
        pBuffer[nBufferIndex].setOutT = m_Inputs.setAussenT;
        pBuffer[nBufferIndex].setRoomT1 = m_Inputs.setRaumT1;
        pBuffer[nBufferIndex].setRoomT2 = m_Inputs.setRaumT2;
        pBuffer[nBufferIndex].setWaterT = m_Inputs.setWasserT;
        pBuffer[nBufferIndex].watertemp = m_Inputs.Twasser;

        ++nBufferIndex;
        if (nBufferIndex == nBuffer)
        {
            nBufferIndex = 0;
        }
        return TRUE;
```

```
    }
    else
    {
        return FALSE;
    }
}


////////////////////////////////////////////////////////////////////////
HRESULT CTcAsyncBufferWritingModule::AddModuleToCaller()
{
    HRESULT hr = S_OK;
    if ( m_spCyclicCaller.HasOID() )
    {
        if ( SUCCEEDED_DBG(hr = m_spSrv->TcQuerySmartObjectInterface(m_spCyclicCaller)) )
        {
            if ( FAILED(hr = m_spCyclicCaller->AddModule(m_spCyclicCaller, THIS_CAST(ITcCyclic))) )
            {
                m_spCyclicCaller = NULL;
            }
        }
    }
    else
    {
        hr = ADS_E_INVALIDOBJID;
        SUCCEEDED_DBGT(hr, "Invalid OID specified for caller task");
    }
    return hr;
}

////////////////////////////////////////////////////////////////////////
VOID CTcAsyncBufferWritingModule::RemoveModuleFromCaller()
{
    if ( m_spCyclicCaller )
    {
        m_spCyclicCaller->RemoveModule(m_spCyclicCaller);
    }
    m_spCyclicCaller    = NULL;
}
```