# BNCS Class Library

CSI Driver Library

Written by Tim Alden



## Overview

This class provides all the required connectivity between your driver application and CSI. It encapsulates all data transfer to and from CSI, and provides the required callback mechanism for asynchronous network messages and status information.

The class definition is as follows:

```
class extdriver {
        HWND hWndSpawn;
        PULONG txcount;
        PULONG rxcount;
        LPSTR netmsgtok;
        UINT iStatus;
        UINT iDrvType;
        static HINSTANCE hInstEx;
        ULONG deftx,defrx;
public:
        UINT iDevice;
        LPSTR netmsg;
        UINT paramcount;
        LPSTR param[16];
        LRESULT (*func)(extdriver*, LPCSTR);
        UINT connect(UINT iDevice, HINSTANCE hInstApp=0, UINT iType=DRVTYPE_NONE);
        UINT connectx(UINT iDevice);
        extdriver();
        ~extdriver();
        void notify(LRESULT(*)(extdriver*, LPCSTR));
        void setcounters(PULONG txcounter, PULONG rxcounter);
        void incrx(void);
        UINT getstate(void);
        UINT setstate(UINT stateflags);
        LRESULT txrevmsg(LPSTR netmessage, UINT iCallingWS);
        LRESULT txquerymsg(LPSTR netmessage, UINT iCallingWS, BOOL lastflag=FALSE);
        void getdbname(WORD device, WORD database, WORD index, LPSTR namebuffer, int iMaxSize=32);
        int getdbindex(WORD device, WORD database, LPCSTR name);
        UINT gettype(void);
        LRESULT setifmode(WORD mode);
        WORD getifmode(void);
        UINT tokenize(void);
};
```

### *Required files*

The class is contained in the library **`csidrv.lib`** and a debug version in **`csidrvdb.lib`**

The header files **`<csidrv.h>`** and **`<bncsdef.h>`** should be referenced by your application.

The following constants are defined in the <bncsdef.h> header file, and are valid values for the return value from calling getstate():

```
#define ERROR_GENERAL               0xFFFF
#define ERROR_WRONG_TYPE            0xFFFE
#define ERROR_BAD_PARAMLIST         0xFFFD

#define COMMAND                     0
#define STATUS                      1
#define RXONLY                      2
#define DISCONNECTED                3
#define INVALID_DRIVERNUM           4
#define CANT_FIND_CSI               5
#define TO_TXRX                     6
#define TO_RXONLY                   7
#define QUERY_TXRX                  8
#define CANT_REGISTER_CLASSWND      9
#define DRIVERNUM_ALREADY           10
#define CSI_NOT_ENABLED             11
#define DATABASECHANGE              12
#define BAD_WS                      13
#define CONNECTED                   15
```

### *Usage*

*Declare an instance of the class extdriver:*

```
extdriver ed;
```

*followed by*

```
ed.connect(iDriverNum[,hInst]);
```

*or*

```
ed.connect(iDriverNum,hInst[,iType]);
```

*The status/success of the connection to CSI can be ascertained by checking the return value. See table below describing the getstate() function.*
*The hInst parameter is the application instance handle, or NULL.*

The following constants are defined in the <bncsdef.h> header file, and are valid values for the *iType* optional 3rd parameter for the above function:

```
#define DRVTYPE_NONE                0xF000  // the default - no filtering
#define DRVTYPE_R                   0x1100
#define DRVTYPE_G                   0x2200
#define DRVTYPE_I                   0x4400
#define DRVTYPE_D                   0x8800
```

*Setting **iType** to a specific driver type (e.g. Router Driver is DRVTYPE_R) will pre-filter commands which are not appropriate for the chosen driver type. In this example, commands beginning with "R" will be passed through, and commands beginning with "I" or "G" will be discarded and an error condition notified.*

*Additional devices can be registered on the same driver connection by using:*

```
ed.connectx(iDriverNumExtra);
```

*The status/success of the connection to CSI can be ascertained by calling the getstate() function.*
*The valid return values are:*

| | |
|---|---|
| CONNECTED | Driver registered correctly – OK to proceed |
| CANT_FIND_CSI | CSI not found |

| | |
|---|---|
| CSI_NOT_ENABLED | CSI not enabled in driver mode |
| DRIVERNUM_ALREADY | The requested driver number is already registered on this workstation |
| INVALID_DRIVERNUM | The requested driver number is not in the valid range |

examples:

```
if (ed.getstate()==CANT_FIND_CSI)
      Debug("CSI not found");
```

*All values other than CONNECTED are fatal errors, and your application should terminate.*

*Assign the callback function which the class will send messages to:*

```
ed.notify(CSINotify);
```

*where the function is of the form:*

**LRESULT CSINotify(extdriver* ped, LPCSTR szMessage);**

*Set up automatic counters for incoming and outgoing messages as follows:*

```
ed.setcounters(&txcounter,&rxcounter);
```

*txcounter and rxcounter should be long integers. The class will automatically increment the value of these variables. The programmer must reset them to zero and display them on the screen as appropriate.*

## Additional parsing feature

*The LPCSTR message supplied to the callback function above is automatically split into elements and stored in the class. These elements can be accessed as follows:*

*Obtain the number of parameters found:*

```
UINT iParamCount=ed.paramcount;
```

*The LPCSTR pointer array* **param** *points to the split elements:*

*Example:*

command "RC 8 3 6 200"

*this means route source 3 to dest 6 on device 8, command from workstation 200*

```
//iParamCount=5 from ped->paramcount        // there are five parameters here
LPSTR szParamCmd=ped->param[0];             // will be "RC"
LPSTR szDevNum=ped->param[1];               // will be "8"
LPSTR szSrc=ed->param[2];                   // will be "3"
LPSTR szDest=ped->param[3];                 // will be "6"
LPSTR szWks=ped->param[4];                  // will be "200"
```

*ped->param[5+] are undefined in this case*

*The programmer can convert any numeric elements to integers using, e.g. atoi()*

```
int iDevNum=atoi(ped->param[1]);            // value of iDevNum will be 8 in example
```

## Sample Callback Function:

*The nature of the incoming message and connection status to CSI can be ascertained by calling the getstate() function.*
*The valid values are:*

| | |
|---|---|
| COMMAND | This callback contains a valid network command |
| STATUS | This callback contains a status message |
| DISCONNECTED | CSI is closing down |
| RXONLY | The driver you want to be is currently running on another workstation in TX/RX mode. You are now in RX only mode |
| TO_RXONLY | A driver on another workstation wants to go TXRX. Return TRUE to yield (you go RX Only) or FALSE to refuse (retain TXRX state) |
| TO_TXRX | A driver on another workstation is closing down or yielding TXRX. Return TRUE to go TXRX, or FALSE to stay RX Only |
| QUERY_TXRX | CSI checking to see if this driver is capable of going TXRX. Return TRUE for yes, FALSE for no. |
| ERROR_WRONG_TYPE | A message was received but was ignored because it is not for this type of driver |
| ERROR_GENERAL | Unspecified error condition |

```
LRESULT Notify(extdriver* ped,LPCSTR szMsg)
{
UINT i;

switch (ped->getstate())
	{
	case COMMAND:
		Debug("Command - state = %d, message is %s",ped->getstate(),szMsg);
		for (i=0;i<ped->paramcount;i++)
			{
			Debug("Parameter %d is %s",i+1,ped->param[i]);
			}
		// process commands
		break;

	case STATUS:
		Debug("Status - state = %d, message is %s",ped->getstate(),szMsg);
		for (i=0;i<ped->paramcount;i++)
			{
			Debug("Parameter %d is %s",i+1,ped->param[i]);
			}
		break;

	case DISCONNECTED:
		DestroyWindow(hWndMain); // terminate application ?
		break;

	case RXONLY:
		Debug (szMsg); // notification that another driver is TXRX (you are RX only)
		break;

	case TO_RXONLY:
		return FALSE; // don't yield to RX Only

	case TO_TXRX:
		break; // go TXRX if you can

	case QUERY_TXRX:
		break; // confirm you are prepared to go TXRX
	}

return TRUE;
}
```

Always check getstate() first in case the callback is informing you of a termination, or other special event. A valid command will always have COMMAND status!

## Revertive Transmission Functions

The following functions can be used to send revertives to CSI in response to poll/query requests or tally changes from controlled devices:

*To send a response to a poll command:*

```
ed.txrevmsg(szRevertiveMessage, iDestWorkstation);
```

*Here the **szRevertiveMessage** is a formatted string containing a revertive message such as "RR 8 6 2" which means "router 8: destination 6 has changed to source 2"*
*The **iDestWorkstation** member should be set to the workstation number of the calling workstation if this is a response to a poll command.*
*If a revertive is a result of a tally change from a controlled device, the **iDestWorkstation** parameter should be set to zero so that all workstations receive the change message.*

*To send a response to a query command:*

```
ed.txquerymsg(szRevertiveMessage, iDestWorkstation, fLast);
```

*Here the **szRevertiveMessage** is a formatted string containing a revertive message as above. The **iDestWorkstation** parameter must be the number of the workstation initiating the query. The **fLast** parameter should be set to FALSE for all but the final revertive message when a range of revertives is sent in response to a query. The final one must have fLast=TRUE to initiate the transfer of revertives to the requesting workstation.*

## Additional Specialist functions

The following functions are implemented to control the dynamics of the driver's operation mode:

The mode constants are found in the <bncs.h> header file and are defined as follows:

```
#define IFMODE_NONE         0
#define IFMODE_RXONLY       1
#define IFMODE_TXRX         2
#define IFMODE_FORCETXRX    3
#define IFMODE_TXRXINQ      4
```

*Set the mode of the driver as follows:*

```
ed.setifmode(iMode);
```

The return value is the new mode

*Read the current mode of the driver as follows:*

```
iMode=ed.getifmode();
```

## Database Functions

The following functions access the database functions of CSI:

*Retrieve a database name from device **iDevice** database **iDbase** for entry **iIdx** and place the result in buffer **szName** with a maximum length of **iMax** characters:*

```
ed.getdbname(iDevice,iDBase,iIdx,szName,iMax);
```

*Retrieve the entry number(index) for device **iDevice** database **iDbase** whose name is **szName:***

```
iIndex=ed.getdbindex(iDevice,iDBase,szName);
```

**Tim Alden**
**12 Nov 2003**