BNCS Class Library

CSI Client Library

Written by Tim Alden

Overview

This class provides all the required connectivity between your client 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 extclient {
        HWND hWndSpawn;
        PULONG txcount;
        PULONG rxcount;
        LPSTR netmsgtok;
        UINT iStatus;
        LPSTR netmsg;
        UINT paramcount;
        LPSTR param[16];
        static HINSTANCE hInstEx;
        ULONG deftx,defrx;
        UINT iWorkstation;
        static LRESULT WINAPI ClientWndProc(HWND, UINT, WPARAM, LPARAM);
public:
        UINT getparamcount(void);
        LPSTR getparam(UINT);
        LRESULT (*func)(extclient*,LPCSTR);
        UINT connect(HINSTANCE hInstApp);
        extclient();
        ~extclient();
        UINT setstate(UINT iState);
        UINT getstate(void);
        void setcachestate(UINT iState);
        UINT getcachestate(void);
        BOOL cachetest(UINT iDevice, UINT iMin, UINT iMax);
        void cacheflush(UINT iDevice);
        LRESULT txrtrcmd(LPCSTR szMessage, BOOL fWhen=LATER);
        LRESULT txinfocmd(LPCSTR szMessage, BOOL fWhen=LATER);
        LRESULT txgpicmd(LPCSTR szMessage, BOOL fWhen=LATER);
        LRESULT regtallyrange(UINT iDevice, UINT iMin, UINT iMax, BOOL fInsert=OVERWRITE);
        LRESULT unregtallyrange(UINT iDevice=0, UINT iMin=0, UINT iMax=0);
        void notify(LRESULT(*)(extclient*,LPCSTR));
        void setcounters(PULONG rxcounter, PULONG txcounter);
        void incrx(void);
        void getdbname(WORD device, WORD database, WORD index, LPSTR namebuffer, int iMaxSize=32);
        int getdbindex(WORD device, WORD database, LPCSTR name);
        void setdbname(WORD device, WORD database, WORD index, LPCSTR name, BOOL fPoll=0);
        UINT tokenize(void);
        LRESULT txnetcmd(LPCSTR szMessage);
};
```

Required files

The class is contained in the library **csicln.lib** and a debug version in **csiclndb.lib**

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

Usage

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 DISCONNECTED                3
#define CANT_FIND_CSI               5
#define CANT_REGISTER_CLASSWND      9
#define DATABASECHANGE              12
#define BAD_WS                      13
#define CONNECTED                   15
```

Declare a (global) pointer to the class extclient:

```
extclient *gpec;
```

Dynamically allocate an instance of the class, then call the connect function, specifying the instance handle to your application:

```
gpec=new extclient;
gpec->connect(hInst);
```

The status/success of the connection to CSI is provided by the return value to the connect() function, or 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
BAD_WS	An invalid or missing workstation number in CSI.INI
CANT_REGISTER_CLASSWND	Cannot register CSI messaging window. Suspect hInst parameter.

example:

```
if (gpec->getstate()==CANT_FIND_CSI)
      Debug("CSI not found");
```

Values other than CONNECTED are fatal errors, and your application should terminate.

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

```
gpec->notify(CSINotify);
```

the function is of the form:

```
LRESULT Notify(extclient* pec, LPCSTR szMessage);
```

Register for device revertives:

```
gpec->regtallyrange(1,1,32[,OVERWRITE]);
```

This registers for device 1, index 1 thru 32. The 4th parameter is optional (default is OVERWRITE)

You can explicity add partial registrations as follows:

```
gpec->regtallyrange(1,39,40,INSERT);
```

This adds registration for index 39 thru 40 to any existing registration for device 1.

Registrations can be deleted as follows:

```
gpec->unregtallyrange(1,17,32);
```

This removes registration for device 1 range 17 thru 32.
All parameters are optional. If only the first parameter is specified, the entire range for that device will be unregistered. If no parameters are specified, all devices will be unregistered.

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

```
gpec->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 screen as appropriate.

Close the CSI client connection by deleting the object:

```
delete gpec;
```

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=gpec->paramcount;
```

The LPCSTR pointer array **param** *points to the split elements, and you can access the array values using the* getparam *public function:*

Example:

revertive message “IR 001 001 001 ‘123’”
this means slot 1 on infodriver 1 now contains text ‘123’

```
//iParamCount=5 from gpec->paramcount              // there are five parameters here
LPSTR szParamRev=gpec->getparam(0);                // will be IR
LPSTR szDevNum=gpec->getparam(1);                  // will be 001
LPSTR szSlot=gpec->getparam(2);                    // will be 001
LPSTR szIndex=gpec->getparam(3);                   // will be 001
LPSTR szContent=gpec->getparam(4);                 // will be ’123’
```

gpec->getparam[5+) are undefined in this case

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

```
int iDevNum=atoi(gpec->getparam(1));        // value of iDevNum will be 1 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:

REVTYPE_R	This callback contains a router revertive
REVTYPE_I	This callback contains an infodriver revertive
REVTYPE_G	This callback contains a GPI revertive
DATABASECHANGE	This callback contains a database change notification
STATUS	This callback contains a status message
DISCONNECTED	CSI is closing down

```
LRESULT CSINotify(extclient* pec,LPCSTR szMsg)
{
UINT i;

switch (pec->getstate())
        {
        case REVTYPE_R:
        case REVTYPE_I:
        case REVTYPE_G:
                Debug ("Revertive message %s",szMsg);
                for (i=0;i<pec->getparamcount();Debug("Param %d is %s",(i++)+1,pec->getparam(i)));
                //TODO: Add parsing function for network commands
                break;

        case DATABASECHANGE:
                Debug ("Database change message - %s",szMsg);
                for (i=0;i<pec->getparamcount();Debug("Param %d is %s",(i++)+1,pec->getparam(i)));
                break;

        case STATUS:
                Debug("Status message is %s",szMsg);
                break;

        case DISCONNECTED:
                Debug("CSI has closed down");
                DestroyWindow(hWndMain);
                break;

        }

return TRUE;
}
```

Notes:
1. *The callback should return TRUE otherwise CSI will delete your client registration.*
2. *Always check the getstate() first in case the callback is informing you of a disconnection/closedown. A valid revertive will always be REVTYPE_x.*

Sending Commands to Drivers

The following functions can be used to send commands to CSI to pass onto the network for transmission to BNCS driver applications:

Examples:

```
gpec->txinfocmd("IW 1 'Slot 8' 8");
```

This sends an IW command to driver 1, setting slot 8 to 'Slot 8'.

```
gpec->txgpicmd("GP 13 1 8");
```

This sends a GP command to driver 13, requesting a poll of GPIs 1-8.

```
gpec->txrtrcmd("RC 28 1 8");
```

This sends a RC command to driver 28, switching source 1 to destination 8.

```
gpec->txnetcmd("NR 123");
```

This sends a NR command to restart workstation 123. Any of the N series messages can be sent so be carefull on how you use this.

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:*

```
gpec->getdbname(iDevice,iDBase,iIdx,szName,iMax);
```

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

```
iIndex=gpec->getdbindex(iDevice,iDBase,szName);
```

*Change the database name for device **iDevice** database **iDbase** index **iIdx** to **szName:***

```
gpec->setdbname(iDevice,iDBase,iIdx,szName);
```

Cache Functions

The following functions manipulate and monitor the CSI cache:

Turn the CSI Cache on or off:

```
gpec->setcachestate(iState);
```

The cache will be turned on if iState=1, or turned off if iState=0.

Find out the current status of the cache:

```
iCurrState=gpec->getcachestate();
```

The current state will be returned in iCurrState. Values as for setcachestate().

*Find out the validity of the cache for device **iDevice** between index **iMin** and **iMax**:*

```
fStatus=gpec->cachetest(iDevice, iMin, iMax);
```

The function returns TRUE or FALSE.

*Flush (clear) the cache for device **iDevice**:*

```
gpec->cacheflush(iDevice);
```

The function will empty the cache of all entries for the specified device. Subsequent polling or queries for that device will result in network messages being sent to the appropriate device rather than being serviced from the cache.