Remote control of PG4UW, user's manual

Version: 1.39
Date: Feb 16th 2017

Preface

There exist more reason why to control the programmer not by PG4UW software directly, but from other software. Therefore we provide a remote controlling capability of PG4UW software.

The remote control feature allows to be PG4UW software flow controlled by other application – either using BAT file commands or using DLL file. This manual describes basic features of remote control capability of PG4UW control program and also explain how to implement the remote control functions available in the PG4UW *remotelb.dll* library file.

Table of Contents

1. Basic facts about remote control 3

2. Published functions in library remotelb.dll 4

2.1 General functions for Client/Server remote control connection establishing and connection parameters setting..... 4

2.2 Procedures used for sending basic types of commands from remote Server to PG4UW (Client)..... 8

2.3 Constants used by remote control communication..... 23

3. Short example of remote control implementation 24

3.1 Synchronous mode..... 24

3.2 Asynchronous mode..... 24

4. Remote control examples - source files 25

4.1 Source files written in Borland Delphi Pascal..... 25

4.2 Source files written C++ (suitable for Borland C++ Builder and Microsoft Visual C++). 25

4.3 Source files written in Microsoft Visual Basic 6..... 25

4.4 Source files written in Microsoft Visual Basic 2002/2003/2005 .NET..... 25

5. Using remote control with multiply programmers (multiprogramming) 26

5.1 General functions for Client/Server remote control connection establishing and connection parameters setting..... 27

5.2 Procedures used for sending basic types of commands from remote Server to PG4UW (Client)..... 28

6. Remote command line control of PG4UW 43

6.1 Configuration command line parameters..... 43

6.2 Executive command line parameters..... 44

6.3 Return values of PG4UWcmd.exe..... 44

6.4 Basic rules for using of executive command line parameters..... 46

7. Remote control of PG4UW applications running on different computer(s) 48

7.1 Sequence of steps is recommended when starting multiprogramming system..... 50

7.2 Basic description of Multidemo application..... 50

8. .NET support 52

9. System requirements 53

9.1 System requirements for remote control program..... 53

9.2 System requirements for application PG4UW when using remote control function..... 53

9.3 Common requirements..... 53

10. Version history of this documentation 55

1. Basic facts about remote control

Remote control of PG4UW control program allows to control some functions of PG4UW application by other application. This is very suitable feature for integrating programmer to mass-production handler system or other useful application.

Remote control main features are:

1. Remote control philosophy is:
   - remote application that controls PG4UW acts as Server
   - PG4UW acts as Client
2. the communication between PG4UW and remote Server is realized by set of commands available in DLL library remotelb.dll
3. communication is asynchronous and it uses events to handle received messages from PG4UW
4. the order of starting PG4UW and Server application is not important but better way is to start the Server application as the first and PG4UW (Client) the second
5. communication between PG4UW and remote control program is made via TCP protocol - this allows the PG4UW to be installed on one computer and remote control application to be installed on another computer, and these computers will be connected together via network
6. remote control library remotelb.dll is written in Borland® Delphi Pascal and is usable by Borland® Delphi Pascal, many C/C++ environments, and also environments with other programming languages, for example Microsoft Visual Basic 6.

Default TCP communication settings for remote control are:

Port: **telnet** Address: **127.0.0.1** or **localhost**

Address setting applies for PG4UW (Client) only. Port setting applies for PG4UW (Client) and also for Server application.

Default settings allows to use remote control on one computer (address localhost). PG4UW (Client) and remote control Server have to be installed on the same computer.

Note: If **firewall** is installed on system, firewall can display warning message when remote control Server or Client is starting. When firewall is showing warning with question asking to allow or deny network access for remote Server or Client, please select 'Allow' option, otherwise remote control will not work. Of course you can specify in firewall options more strict rights to allow remote Server/Client access on specified address and port only.

2. Published functions in library remotelb.dll

Following description presents purpose of each function. For more particular description of function declarations and parameters, please see the file RemoteCtrl.pas.

If the remote Server application is written in C language, there is necessary to write appropriate header .h file to make functions from library remotelb.dll available in C/C++ project.
The Pascal unit file RemoteCtrl.pas shows function declarations and parameters. This can be used for writing C/C++ header file.

2.1 General functions for Client/Server remote control connection establishing and connection parameters setting

To see the usage of following functions and procedures, see the example application PG4UWrem, especially Pascal unit remoteform.pas.

```
procedure CreateClientAndMakeConnectionToServer( vProcessProc: TProcWithPChar;
                                                 vWriteToLogProc: TProcWithPChar;
                                                 vPort, vAddr: PChar); stdcall;
```

Procedure creates Client communication object, with defined parameters and tries to connect to Server. This procedure is used internally in PG4UW (Client) application. Server application should not use this procedure.

```
procedure CreateServerAndMakeListenToClients( vProcessProc: TProcWithPChar;
                                              vWriteToLogProc: TProcWithPChar;
                                              onClientConnectProc: TProc;
                                              onClientDisconnectProc: TProc;
                                              vPort: PChar); stdcall;
```

Procedure creates Server communication object, with defined parameters and starts waiting for Client(s). This procedure is used by Server application - remote control program.

Input parameters are:

vProcessProc: TProcWithPChar - define pointer to optional callback procedure which will be called every time Server receives message(s) from (PG4UW) Client. This is usual way to receive information or commands from remote PG4UW Client.
If 'vProcessProc' is defined as **nil**, no “on-client command received“ event will be used. Then the only way how to receive information and commands from Client is reading of commands explicitly by calling of special function 'GetCommandStringFromFIFO'. For more details please take a look at description of function GetCommandStringFromFIFO.

vWriteToLogProc: TProcWithPChar - define pointer to optional callback procedure which is useful especially during debugging of Server program. Procedure is called when any of basic Client-Server communication events will occure.
Communication events are: connect/disconnect Client-Server, send message to Client, receive message from Client. Procedure can contain user defined write to memo or log window of Server application. If it is defined as **nil**, no “write-to-log“ events will be used.

onClientConnectProc: TProc - optional callback procedure is called as event when Client is connected to Server. If it is defined as **nil**, no “on-client connect“ event will be used.

onClientDisconnectProc: TProc - optional callback procedure is called as event when Client is disconnected from Server. If it is defined as **nil**, no “on-client disconnect“ event will be used.

vPort: PChar - defines port for TCP communication (default is 'telnet')

Server does not have address defined itself. Internally Server has defined address 0.0.0.0, which means, that Server accepts Clients from all interfaces.

Code example (callback method):

```
procedure intServerProcessProc(CMDline: ansistring);
// procedure handles one line of message from client
var
        tmpstr: string;
        index, q: integer;
begin
        CMDline := trim(CMDline);
        // now try to obtain client index from CMDline (not used in single programming mode)
        if pos(TCP_KEYWORD_CINDEX, LowerCase(CMDline)) = 1 then
        begin
                tmpstr := trim(copy(CMDline, length(TCP_KEYWORD_CINDEX)+1, length(CMDline)));
                tmpstr := trim(copy(tmpstr, 1, pos('|', tmpstr)-1));
                if not Str2int(tmpstr, index) then exit;
                q := pos('|', CMDline);
                if q > 0 then CMDline := trim(copy(CMDline, q+1, length(CMDline)))
                else CMDline := '';
        end; { if pos(TCP_KEYWORD_CINDEX, LowerCase(CMDline)) = 1... }

        {CMD program is busy}
        if CompareText(CMDline, TCP_CMD_PROG_IS_BUSY) = 0 then
                Process_CMD_PROG_IS_BUSY

        {..... etc.   (processing of other commands) }
end;

procedure ServerProcessProc(CMDline: PChar); stdcall;
{ This procedure is used like callback procedure.
it handles one or more lines of messages from client by sequential
calls of intServerProcessProc. Each line of string represents one message.
More lines in string mean more commands. }
const
        CRLF = #13#10;
var
        tmpline, tmpstr: ansistring;
        q: integer;
begin
        tmpline := CMDline;
        // more commands can be cumulated into one packet, each command is at one line
        repeat
                q := Pos(CRLF, tmpline);
                if q > 0 then
                begin
                        tmpstr := copy(tmpline, 1, q-1);
                        delete(tmpline, 1, q+length(CRLF)-1);
                end
                else
                        tmpstr := tmpline;
                if tmpstr <> '' then intServerProcessProc(tmpstr);
        until q = 0;
end; { procedure }

CreateServerAndMakeListenToClients(ServerProcessProc,// CallbackProcPtr
                                   nil,               // WriteToLogwindowProc
                                   nil,               // onClientConnectProc
                                   nil,               // onClientdisconnectProc
                                   PChar(remote_ctrl_settings.Client_Server_Port));
```

Code example (non-callback method):

```
procedure intServerProcessProc(CMDline: ansistring);
// procedure handles one line of message from client
var
        tmpstr: string;
        index, q: integer;
begin
        CMDline := trim(CMDline);
        // now try to obtain client index from CMDline (not used in single programming mode)
        if pos(TCP_KEYWORD_CINDEX, LowerCase(CMDline)) = 1 then
        begin
                tmpstr := trim(copy(CMDline, length(TCP_KEYWORD_CINDEX)+1, length(CMDline)));
                tmpstr := trim(copy(tmpstr, 1, pos('|', tmpstr)-1));
                if not Str2int(tmpstr, index) then exit;
                q := pos('|', CMDline);
                if q > 0 then CMDline := trim(copy(CMDline, q+1, length(CMDline)))
                else CMDline := '';
        end; { if pos(TCP_KEYWORD_CINDEX, LowerCase(CMDline)) = 1... }

        {CMD program is busy}
        if CompareText(CMDline, TCP_CMD_PROG_IS_BUSY) = 0 then
                Process_CMD_PROG_IS_BUSY

        {..... etc.   (processing of other commands) }
end;

procedure ProcessProcByFIFO;
{ This procedure is used for receiving info and commands from CLient PG4UW app.
The procedure can be used when no callback procedure is defined in calling
of procedure CreateServerAndMakeListenToClients. Callback proc is not defined
by setting its value to nil. }
const
        MAX_ITERATIONS = 3;
        proc_iterations: integer = 0;
var
        received_cmd: PChar;
begin
        if proc_iterations > MAX_ITERATIONS then exit;
        inc(proc_iterations);
        try
        // get command from FIFO
        received_cmd := GetCommandStringFromFIFO;
        while received_cmd <> nil do begin
                // if there was command received from FIFO, call the process proc to process the command
                intServerProcessProc(received_cmd);
                // try to get next command from FIFO
                received_cmd := GetCommandStringFromFIFO;
        end; { while }
        finally
                if proc_iterations > 0 then dec(proc_iterations);
        end; { try }
end; { procedure }

CreateServerAndMakeListenToClients(nil,   // CallbackProcPtr
                                   nil,   // WriteToLogwindowProc
                                   nil,   // onClientConnectProc
                                   nil,   // onClientdisconnectProc
                                   PChar(remote_ctrl_settings.Client_Server_Port));
```

procedure **MakeClientConnectionToServer**(FailedErrDisplay: boolean); stdcall;

Procedure tries to connect Client to Server. The procedure is used internally in PG4UW (Client) application. Server application should not use this procedure.

procedure **MakeClientDisconnectionFromServer**; stdcall;

Procedure tries to disconnect Client from Server. The procedure is used internally in PG4UW (Client) application. Server application should not use this procedure.

procedure **MakeClientServertCloseConnectionAndFree**; stdcall;

Procedure is used for Client and Server applications to close connection and free TCP Client/Server communication object.
The Client and Server applications call this routine automatically when they closed.

procedure **SendOperationResultToServer**(op_result: TOpResultForRemote); stdcall;

Procedure is used by Client applications for sending messages to Server. Server application should not use this procedure.

procedure **SendLineToServer**(line: PChar); stdcall;

Procedure is used by Client applications for sending messages to Server. Server application should not use this procedure.

function **ClientServerObjectExists**: boolean; stdcall;

Function returns true, if Client/Server object already exists, otherwise returns false. Function does not test connection status.

function **ClientConnectionIsClosed**: boolean; stdcall;

Function returns true, if Client connection status is 'Closed', otherwise returns false. Function is used by PG4UW (Client) application. Server application should not use this procedure.

function **ServerHasConnectedClient**: boolean; stdcall;

Function is used by Server application. Function returns true, if Client is connected to Server, otherwise returns false.

procedure **EnableWriteEventsToLog**(value: boolean); stdcall;

Procedure is used by Client and Server applications. The purpose of the procedure is to block calls of procedure vWriteToLogProc defined as parameter of procedures CreateClientAndMakeConnectionToServer and CreateServerAndMakeListenToClients.

function **GetCurrentPort**: PChar; stdcall;

Function is used by Client and Server applications. Function returns the value of current port (for example '2020', 'telnet', etc.).

function **GetCurrentAddr**: PChar; stdcall;

Function is used by Client applications. Function returns the value of current address (for example 'localhost' or '127.0.0.1', '192.168.0.10', ...).
For Server, the function returns always value '0.0.0.0'.

procedure **ClientServerProcessMessages**; stdcall;

Procedure is used to quickly build a working message pump. It loops through message processing until all messages are processed. This function is very similar to TApplication.ProcessMessage(). The procedure is suitable especially if your application has no TApplication object (Forms unit not referenced at all).

2.2 Procedures used for sending basic types of commands from remote Server to PG4UW (Client)

procedure **SEND_CMD_BringToFront**; stdcall;

Procedure is used to send message 'bringtofront' to Client. When Client receives the message, it tries to make BringToFront operation. BringToFront operation is basically activation of Client application window.

Example:

Server request:

SEND_CMD_BringToFront;

Answer from client:

- client send no answer

procedure **SEND_CMD_ShowMainForm;** stdcall;

Procedure is used to send message 'showmainform' to Client. When Client receives the message, it makes Show command for main window of the Client application. The message does nothing if the main window of Client application is already visible. Please use this command only in situations when no operation is currently running in the Client application.

Example:

Server request:

SEND_CMD_ShowMainForm;

Answer from client:

- client send no answer

procedure **SEND_CMD_HideMainForm;** stdcall;

Procedure is used to send message 'hidemainform' to Client. When Client receives the message, it makes Hide command for main window of the Client application. Also Client's icon placed on taskbar will be hidden and small icon appears on the tray panel. The message does nothing if the main window of Client application is already hidden. Please use this command only in situations when no operation is currently running in the Client application.

Example:

Server request:

SEND_CMD_HideMainForm;

Answer from client:

- client send no answer

procedure **SEND_CMD_CloseApp;** stdcall;

Procedure is used to send message 'closeapp' to Client. When Client receives the message, it makes Close command for main window of the Clients application. The Close command can be properly performed when no operation is currently running in the Client application. Please use the SEND_CMD_CloseApp command when no operation on device is running and no modal dialogs are shown. To ensure that no operation is currently running, command **SEND_CMD_GetProgStatus** can be used. Also commands **SEND_CMD_Stop** for operation stopping can be used.
For more details see enclosed examples of remote control written in Pascal and C++ languages. Examples are described in the chapter III. of this manual.

Example 1:

Server request:

```
SEND_CMD_CloseApp;
```

Answer from client:

- client send no answer

Example 2:

```
procedure Tformsimpledemo.btnDisconnectProgClick(Sender: TObject);
begin
    if not ServerHasConnectedClient then
        btnConnectProg.enabled := true
    else
        // check if PG4UW app is not busy
        if device_progress.busy then
        begin
            {if} application.MessageBox(Pchar('Programmer is still busy.'#13#10#13#10 +
            'Please wait to complete current operation on running programmer or use Stop command'#13#10 +
            'to stop current operation'),
            PChar(caption),
            mb_ok + MB_ICONWARNING); exit;
        end { if device_progress[index].busy... }
        else begin
        // the first step is to send commands Stop operations - two times
        SEND_CMD_Stop;
        Application.ProcessMessages;
        Sleep(250); {wait}
        SEND_CMD_Stop;
        Application.ProcessMessages;
        Sleep(250); {wait}
        // the next step is to send commands Close aplication
        SEND_CMD_CloseApp;
        Application.ProcessMessages;
        Sleep(250); {wait}
        btnConnectProg.enabled := true;
    end; { if device_progress.busy... }

    // if Sender is "Close" button, let's close remote control application
    disconnect_was_Successful := true;
end; { procedure }
```

procedure **SEND_CMD_BlankCheckDevice**; stdcall;

Procedure is used to send message 'blankcheck' to Client. When Client receives the message, it starts device Blank check operation. In the end of operation Client sends operation result to Server. Operation result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.
Client PG4UW can accept and start 'blankcheck' command only when no operation is currently running in PG4UW. If PG4UW is running some operation, the 'blankcheck' command will be

ignored. To receive the current status of programmer, use command SEND_CMD_GetProgStatus described later.
When last device operation differs from currently selected device operation, we have to send command STOP (to close Repeat wait dialog in PG4UW).

Example 1:

Server request:

SEND_CMD_BlankCheckDevice;

Answer from client:

See the Client send operation result chapter.

Example 2:

```
procedure Tformsimpledemo.btnRun_zalClick(Sender: TObject);
const
    lastrunop: TAppOperationType = otNone;
var
    AppOpType: TAppOperationType;
begin
    // check if appication is running and ready to go and when it is ready,
    // allow remote control Start operation command send to PG4UW
    if ServerHasConnectedClient then
    begin
    // check if last project or file load operation was successfull
        if load_file_result <> frgood then
        begin
            application.MessageBox('Operation "Run" can not be performed, because last project or file load
            was not successful.'#13#10#13#10 +
            'Please use button "Select Project file" to select wished project file to load and then use'#13#10 +
            'command button "Load Selected Project file" to make project load.',
            PChar(caption), mb_ok + MB_ICONWARNING);
            exit;
        end; { if load_file_result <> frgood... }
         // check if PG4UW app is not busy
        if device_progress.busy then
        begin
            {if} application.MessageBox(Pchar('Programmer is still busy.'#13#10#13#10 +
            'Please wait to complete current operation on running programmer or use Stop command'#13#10
            + 'to stop current operation'),
            PChar(caption),
            mb_ok + MB_ICONWARNING);
            exit;
        end;
        btnRun.enabled := false;
        btnLoadSelectedPrj.enabled := false;
        btnLoadSelectedFile.enabled := false;

        // check, what device operation has to be started
        AppOpType := MCsettingsRec.app_user_devop_array;

        // when last device operation differs from currently selected device operation,
        // we have to send command STOP (to close Repeat wait dialog in PG4UW)
        if AppOpType <> otRunApp then
            if lastrunop <> AppOpType then
            begin
                lastrunop := AppOpType;
                SEND_CMD_Stop;
                application.processmessages;
                sleep(350);
            end; { if lastrunop <> AppOpType... }

        // now let's send wished command to PG4UW
        case AppOpType of
              otNone: ;
            otRunApp: SEND_CMD_BringToFront;
        otblankCheck: SEND_CMD_BlankCheckDevice;
```

```
                        otReadDevice: SEND_CMD_ReadDevice;
                      otVerifyDevice: SEND_CMD_VerifyDevice;
                   otProgramDevice: SEND_CMD_ProgramDevice;
                      otEraseDevice: SEND_CMD_EraseDevice;
                              otStop: SEND_CMD_Stop;
                else SEND_CMD_BringToFront; // unknown operation (do Bring to front only)
                end; { case }
                sleep(100); {wait}
                application.processmessages;
                sleep(20); {wait}
        end { if GetIfAnyClientAppIsConnectedToServer... }
        else
                application.MessageBox(Pchar('There are no PG4UW applications connected.'#13#10#13#10 +
                                        'Please use the "Connect programmer" button to run'#13#10 +
                                        'and connect PG4UW application.'),
                                        PChar(caption),
                                        mb_ok + MB_ICONWARNING);
  end; { procedure }

//example of procedure for processing client answer
procedure Process_CMD_OPRESULT(CMDline: ansistring);
var
        tmpstr: string;
        btn_run_visible: boolean;
begin
        // enable write commands to log - but if pointer WriteToLogwindowProc in
        // CreateServerAndMakeListenToClients is nil, following line has no effect
        EnableWriteEventsToLog(true);

        tmpstr := copy(CmdLine, length(TCP_CMD_OPRESULT)+1, length(CmdLine));
        // remember device result
        if tmpstr = TCP_KEYWORD_OPRESULT_GOOD then
        begin
                device_op_result := oprGood;
                inc(statist_rec.good);
                inc(statist_rec.total);
        end { if tmpstr = TCP_KEYWORD_OPRESULT_GOOD... }
        else if tmpstr = TCP_KEYWORD_OPRESULT_FAIL then
        begin
                device_op_result := oprFail;
                inc(statist_rec.fail);
                inc(statist_rec.total);
        end { else if tmpstr = TCP_KEYWORD_OPRESULT_FAIL... }
        else if tmpstr = TCP_KEYWORD_OPRESULT_HWERR then
        begin
                device_op_result := oprHWError;
                inc(statist_rec.fail);
                inc(statist_rec.total);
        end { else if tmpstr = TCP_KEYWORD_OPRESULT_HWERR then... }
        else device_op_result := oprNone;
end;
```

procedure **SEND_CMD_ReadDevice**; stdcall;

Procedure is used to send message 'readdevice' to Client. When Client receives the message, it starts device Read operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

SEND_CMD_ReadDevice;

Answer from client:

See the Client send operation result chapter.

procedure **SEND_CMD_VerifyDevice**; stdcall;

Procedure is used to send message 'verifydevice' to Client. When Client receives the message, it starts device Verify operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

SEND_CMD_VerifyDevice;

Answer from client:

See the Client send operation result chapter.

procedure **SEND_CMD_ProgramDevice**; stdcall;

Procedure is used to send message 'programdevice' to Client. When Client receives the message, it starts device Program operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

SEND_CMD_ProgramDevice;

Answer from client:

See the Client send operation result chapter.

procedure **SEND_CMD_EraseDevice**; stdcall;

Procedure is used to send message 'erasedevice' to Client. When Client receives the message, it starts device Erase operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

SEND_CMD_EraseDevice;

Answer from client:

See the Client send operation result chapter.

procedure **SEND_CMD_RepeatLastDevOperation**; stdcall;

Procedure is used to send lastly used device operation command to Client. For example if lastly used command is SEND_CMD_ProgramDevice, the call of procedure SEND_CMD_RepeatLastDevOperation will be the same as call of SEND_CMD_ProgramDevice.

Example:

Server request:

SEND_CMD_RepeatLastDevOperation;

Answer from client:

- client send operation result

procedure **SEND_CMD_Stop**; stdcall;

Procedure is used to send message 'stopoperation' to Client. When Client receives the message, it stops current device operation. If no operation is running, the 'stopoperation' command does nothing. Other function of 'stopoperation' is closing message window(s) in Client application.

Example:

Server request:

SEND_CMD_Stop;

Answer from client:

- client send no answer

procedure **SEND_CMD_SelectDevice**(devmanuf, devname: PChar); stdcall;

Procedure is used to send message 'selectdevice:' to Client. When Client receives the message, it tries to select specified device. Device is specified by parameters devmanuf and devname. Parameters are not case sensitive. The Client sends select device result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

Example: To select device Intel TE28F160C3B [TSOP48] call

Server request:

SEND_CMD_SelectDevice('Intel', 'TE28F160C3B [TSOP48]');

Answer from client:

- client send operation result

```
// codes for operations result
TCP_CMD_SELECT_DEVICE_RESULT   = 'selectdeviceresult:';
```

```
TCP_SELECT_DEVICE_GOOD      = 'good';
TCP_SELECT_DEVICE_ERROR     = 'error';

//answer will be in followed form
TCP_CMD_SELECT_DEVICE_RESULT + TCP_SELECT_DEVICE_GOOD
//or
TCP_CMD_SELECT_DEVICE_RESULT + TCP_SELECT_DEVICE_ERROR
```

procedure **SEND_CMD_EPROMFLASH_AutoSelect**(pinsnumber: PChar); stdcall;

Procedure is used to send message 'autoseldevice:' to Client. When Client receives the message, it starts autoselect device operation. Parameter pinsnumber can be used to specify the pin number of device which helps more reliable detection of device type. If parameter pinsnumber is blank Pchar (''), autoselect operation tries to detect inserted device pin number automatically. The Client sends currently selected device to Server but not result of autoselect detection success.

Example:

Server request:

```
SEND_CMD_EPROMFLASH_AutoSelect(''); //device pins detected automatically

SEND_CMD_EPROMFLASH_AutoSelect('48'); //48-pins device
```

Answer from client:

```
- client send currently selected device

// codes for operations result
TCP_CMD_AUTSEL_EPRFLSH_RESULT  = 'autoseldeviceresult:';

//answer will be in followed form
TCP_CMD_AUTSEL_EPRFLSH_RESULT + 'manufacturer' + '  ' + 'device name' + ' '
```

procedure **SEND_CMD_LoadProject**(prjname: PChar); stdcall;

Procedure is used to send message 'loadproject:' to Client. When Client receives the message, it tries to load project file specified by parameter prjname. The Client sends load project result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

Example:

Server request:

```
SEND_CMD_LoadProject('C:\test.eprj');
```

Answer from client:

```
- client send project load result

// codes for Load file/project result
TCP_CMD_LOAD_FILE_PRJ_RESULT   = 'loadresult:';
TCP_FILE_LOAD_GOOD           = 'frgood';
TCP_FILE_LOAD_ERROR           = 'frerror';

//answer will be in followed form
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_GOOD
//or
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_ERROR
```

procedure **SEND_CMD_GetProjectFileChecksum**(prjname: PChar); stdcall;

Procedure is used to send message 'getprojectfilechecksum:' to Client. When Client receives the message, it tries to compute CRC-32 of project file specified by parameter prjname. The Client sends result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

procedure **SEND_CMD_LoadFile**(fname: PChar); stdcall;

Procedure is used to send message 'loadfile:' to Client. When Client receives the message, it tries to load file specified by parameter fname. The Client sends load file result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

Example:

Server request:

```
SEND_CMD_LoadFile("C:\test.bin');
```

Answer from client:

```
- client send file load result

// codes for Load file/project result
TCP_CMD_LOAD_FILE_PRJ_RESULT   = 'loadresult:';
TCP_FILE_LOAD_GOOD             = 'frgood';
TCP_FILE_LOAD_ERROR            = 'frerror';
TCP_FILE_LOAD_CANCELLED        = 'frcancelled';

//answer will be in followed form
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_GOOD
//or
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_ERROR
//or
TCP_CMD_LOAD_FILE_PRJ_RESULT +  TCP_FILE_LOAD_CANCELLED
```

procedure **SEND_CMD_GetProgStatus**; stdcall;

Procedure is used to send message 'getprogstatus' to Client. When Client receives the message, it sends its current status info to Server. The status info command received from Client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.
Status info contains four basic info status data:
- busy status
- current device operation type
- current device operation progress

For more information see the example unit

Example 1:

Server request:

```
SEND_CMD_GetProgStatus;
```

Answer from client:

- client current status information

```
// operation_code enumeration type
operation_code = (otNone, otRunApp, otblankCheck, otReadDevice, otVerifyDevice, otProgramDevice, otEraseDevice,
                  otPlayPlayer, otLoadProject, otStop, otSelfTest, otSelfTestPlus);

//codes for current status information
TCP_KEYWORD_OPTYPE = 'optype';
TCP_KEYWORD_PROGRESS  = 'progress';
TCP_CMD_PROGRAMMER_READY_STATUS = 'programmerreadystatus:';
TCP_KEY_PROGRAMMER_NOTFOUND = 'notfound';
TCP_KEY_PROGRAMMER_READY    = 'ready';

//answer will be in followed form
TCP_KEYWORD_OPTYPE + ':' + operation_code + ' '  +
TCP_KEYWORD_PROGRESS +' :'  + device_progress + ' '  +
TCP_CMD_PROGRAMMER_READY_STATUS + TCP_KEY_PROGRAMMER_READY
//or
TCP_KEYWORD_OPTYPE + ':' + operation_code + ' '  +
TCP_KEYWORD_PROGRESS +' :'  + device_progress + ' '  +
TCP_CMD_PROGRAMMER_READY_STATUS + TCP_KEY_PROGRAMMER_NOTFOUND
```

Example 2:

send request to client:

```
SEND_CMD_GetProgStatus;
```

processing the response from client:

```
procedure Process_KEYWORD_OPTYPE(CMDline: ansistring);
var
        tmpstr: string;
        progress, errcode, q: integer;
begin
        // find out current operation type
        q := 1; //pos of TCP_KEYWORD_OPTYPE
        tmpstr := copy(CMDline, q + length(TCP_KEYWORD_OPTYPE)+1, length(CMDline));
        q := pos(' ', tmpstr);
        if q > 0 then tmpstr := trim(copy(tmpstr, 1, q-1))
        else tmpstr := 'unknown';

        // set cur op type
        val(tmpstr, q, errcode);
        if errcode = 0 then device_progress.opcode := q
        else device_progress.opcode := 0;

        // find out current operation progress
        q := pos(TCP_KEYWORD_PROGRESS, CMDline);
        if q > 0 then
        begin
                tmpstr := copy(CMDline, q + length(TCP_KEYWORD_PROGRESS)+1, length(CMDline));
                q := pos(' ', tmpstr);
                if q > 0 then tmpstr := trim(copy(tmpstr, 1, q-1))
                else tmpstr := 'unknown';
                val(tmpstr, progress, errcode);
                if errcode = 0 then device_progress.progress := progress
                else device_progress.progress := 0;
        end; { if q > 0... }

        // find out current programmer status - Ready or Not found (Demo mode)
        q := pos(TCP_CMD_PROGRAMMER_READY_STATUS, CMDline);
        if q > 0 then
        begin
                tmpstr := copy(CMDline, q + length(TCP_CMD_PROGRAMMER_READY_STATUS), length(CMDline));
                q := pos(' ', tmpstr);
                if q > 0 then tmpstr := trim(copy(tmpstr, 1, q-1))
                else tmpstr := trim(tmpstr);
                if tmpstr = TCP_KEY_PROGRAMMER_READY then
```

```
                begin
                        device_progress.programmer_present := true;   // programmer is Ready
                        if device_progress.programmer_ready <> programmer_is_busy then
                                device_progress.programmer_ready := programmer_ready_yes;
                end
                else begin
                        device_progress.programmer_present := false; // programmer is in Demo mode
                        device_progress.programmer_ready := programmer_ready_no;
                end;
        end; { if q > 0... }
```

procedure **SEND_CMD_Quest_is_Client_Ready**; stdcall;

Procedure sends question to Client, if the Client is Ready. Client sends ready yes/no status message to Server. The ready status command received from Client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.
Client is ready when no device operation is currently running and main Client's window is not hidden by any modal dialogs in Client.

Example:

Request from server:

```
SEND_CMD_Quest_is_Client_Ready;
```

Answer from client:

```
// codes for server to client 'ready' question
TCP_CMD_CLIENT_READY_ANSWER = 'clienprogramisreadyanswer';
TCP_KEY_CLIENT_READY_YES    = 'isready';
TCP_KEY_CLIENT_READY_NO     = 'isnotready';

//answer will be in followed form
TCP_CMD_CLIENT_READY_ANSWER + TCP_KEY_CLIENT_READY_YES;
//or
TCP_CMD_CLIENT_READY_ANSWER + TCP_KEY_CLIENT_READY_NO;
```

procedure **SEND_CMD_PROCESS_CMDLINE**(cmdparams: PChar); stdcall;

Procedure is used to send message 'cmdlineparams:' to Client. When Client receives the message, it starts processing of received parameters. The procedure is used for internal purposes. Please do not use the SEND_CMD_PROCESS_CMDLINE.

function **SEND_CMD_GetProgrammerConnectedStatus**: BOOL; stdcall;

Function is suitable to detect, if the programmer is successfully connected and present in PG4UW application. Function sends command "get programmer connection status" to client application and waits for client response (timeout is 3 seconds). Function has no parameters.

Return values are:
True - OK - programmers is successfully connected
False - BAD - programmer is not found or no response received (timeout occurs)

function **GetCommandStringFromFIFO**: PChar; stdcall;

Function is used to read next remote received information or control command from connected Client (PG4UW) application. The commands received from client are stored internally in remotelb.dll in FIFO buffer. User can access the FIFO sequentially by this function. Each call of this function will pick the oldest command from FIFO buffer and decrements internal FIFO counter by 1.

If the number of items in FIFO is 0, function will return value *nil*.

Return values are:
If the function succeeds, the return value is pointer to command string read from FIFO.
If the function fails, the return value is nil.
Function fails, if it is called while FIFO buffer of commands is empty or FIFO does not exist.

This function is especially useful, when user does not use callback functions in server remote control application. In such case, the commands received from PG4UW Client can be read by using function

GetCommandStringFromFIFO_CINDEX. More details about using the function, can be seen in remote control examples.

Notes:
1. The function is active, only when callback procedure pointer specified by parameter 'vProcessProc' in remote control initialization procedure "CreateServerAndMakeListenToMultiClients" is nil. If there is specified callback procedure (vProcessProc <> nil), the function GetCommandStringFromFIFO is not usable, and it will always return value nil.

2. The FIFO buffer has size 128 items. It means, it can store maximally of 128 commands received from Client PG4UW. So it is recommended to call function GetCommandStringFromFIFO_CINDEX periodically to pop commands from FIFO and prevent FIFO buffer from getting full. If the number of commands received is greater than capacity of FIFO memory, the newly coming commands are ignored. Of course the Client does not send commands or information to Server application without requiring them from Server. So reading commands from FIFO has sense in cases, the Server sends commands to Client.

3. Commands received by function 'GetCommandStringFromFIFO' are in form of null-terminated strings. The type of null-terminated string is known as:
   - PChar in Pascal language
   - char * in C/C++

   A null-terminated string is a zero-based array of characters that ends with NULL (#0); since the array has no length indicator, the first NULL character marks the end of the string.

function **SEND_CMD_WBUFFER**(bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to write specified block of data to main device buffer of remote client (PG4UW program).

Parameters used are:
bufaddr - buffer address in PG4UW, where data have to be written
bufptr - pointer to memory block of data, which have to be sent to PG4UW
bufsize - number of bytes, which have to be sent to PG4UW, valid values are from 1 to 256 bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_WBUFFER will return error code. If TimeOut is 0, function SEND_CMD_WBUFFER will wait for client response until client is responding for max. time 3 seconds.

Return values are:
0 - OK - data was sent successfuly
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occured, client is not responding

4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_RBUFFER**(bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to read specified region of main device buffer from PG4UW remote client to memory region specified by pointer parameter bufptr.

Parameters used are:
bufaddr - buffer start address in PG4UW, data are read from this address
bufptr - pointer to memory block, where data has to be copied
bufsize - number of bytes, which have to be read from PG4UW buffer, valid values are from 1 to 256 bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_RBUFFER will return error code. If TimeOut is 0, function SEND_CMD_RBUFFER will wait for client response until client is responding for max. time 3 seconds.

Return values are:
0 - OK - data was sent successfully
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occurred, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_WBUFFER_EX**(buffindex: integer;
bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to write specified block of data to specified device buffer of remote client (PG4UW program). The function is very similar to function **SEND_CMD_WBUFFER**. The only difference is one more parameter – buffindex, which is used to specify buffer to be used for writing data.

Parameters used are:
buffindex - specifies the order of buffer, where data will sent, the main buffer has index '1'. The first secondary buffer has index '2', etc. Please note, the secondary buffer(s) is(are) available for some kinds of devices only (e.g. Microchip PIC16F628). The kind of buffer indexed by parameter buffindex depends on order of buffer in application PG4UW in dialog View/Edit buffer. For example device Microchip PIC16F628 has additional buffer with label "Data EEPROM". This buffer can be accessed for data write(s) by this function when buffindex = 2 is specified.
bufaddr - buffer address in PG4UW, where data have to be written
bufptr - pointer to memory block of data, which have to be sent to PG4UW
bufsize - number of bytes, which have to be sent to PG4UW, valid values are from 1 to 256 bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_WBUFFER_EX will return error code. If TimeOut is 0, function SEND_CMD_WBUFFER_EX will wait for client response until client is responding for max. time 3 seconds.

Return values are:
0 - OK - data was sent successfuly
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_RBUFFER_EX**(buffindex: integer;
bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to read specified region of specified device buffer from PG4UW remote client to memory region specified by pointer parameter bufptr. The function is very similar to function **SEND_CMD_RBUFFER**. The only difference is one more parameter – buffindex, which is used to specify buffer to be used for reading data from.

Parameters used are:
buffindex - specifies the order of buffer, where data will sent, the main buffer has index '1'. The first secondary buffer has index '2', etc. Please note, the secondary buffer(s) is(are) available for some kinds of devices only (e.g. Microchip PIC16F628). The kind of buffer indexed by parameter buffindex depends on order of buffer in application PG4UW in dialog View/Edit buffer. For example device Microchip PIC16F628 has additional buffer with label "Data EEPROM". This buffer can be accessed for data write(s) by this function when buffindex = 2 is specified.
bufaddr - buffer start address in PG4UW, from data has to be read
bufptr - pointer to memory block of data, where data has to be copied
bufsize - number of bytes, which have to be read from PG4UW buffer, valid values are from 1 to 256 bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_RBUFFER_EX will return error code. If TimeOut is 0, function SEND_CMD_RBUFFER_EX will wait for client response until client is responding for max. time 3 seconds.

Return values are:
0 - OK - data was sent successfuly
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_SaveFile**(filename: PChar;
fileformat: integer;
timeout: integer): integer; stdcall;

Function is used to write buffer content of remote client (PG4UW) to specified file at specified file format.

Parameters used are:
filename - name of file, where buffer data have to be saved
fileformat - can be one of following values:
FILEFORMAT_BINARY - binary file format
FILEFORMAT_INTELHEX - IntelHex file format

FILEFORMAT_MOTOROLA - Motorola file format
FILEFORMAT_ASCIISPACE - ASCII Space file format
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function **SEND_CMD_SaveFile** will return error code. If TimeOut is 0, function SEND_CMD_SaveFile will wait for client response until client is responding for max. time of 15 seconds

Return values are:
0 - OK - data was saved to file successfully
Error return codes:
1 - unsupported (incorrect) file format
2 - file save error
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - file save error or unknown error

function **SEND_CMD_GetDev_Checksum**(timeout: integer;
pchecksum: LPDWORD): integer; stdcall;

Function sends command “get device checksum” to PG4UW client application. Function is waiting for client's response.

Parameters used are:
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, the function will return error code. If TimeOut is 0, function will wait for client response until client is responding for max. time 10 seconds.

pchecksum - address of variable for checksum value - points to a 32-bit DWORD variable that receives the checksum value

Return values are:
0 - OK - checksum was received successfuly
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

Checksum returned by function *SEND_CMD_GetDev_Checksum* in the variable referenced by pointer *pchecksum* is the same checksum value as checksum displayed in main window of PG4UW control program in table "Addresses". The checksum means sum of current data in main buffer of PG4UW by following rules:

- the checksum is calculated by summing the contents of buffer data from address "Buffer Start" to address "Buffer End". "Buffer Start" and "Buffer End" addresses are displayed in table "Adresses" in the main program window of PG4UW
- the checksum value is displayed in 32-bit hexadecimal number format
- any carry bits exceeding 32-bits are neglected
- buffer data are summed byte-by-byte irrespective of current buffer view mode (x8/x16/x1) organisation

function **SEND_CMD_GET_PG4UW_VERSION**(timeout: integer): PChar; stdcall;

Function sends command “get sw version” to PG4UW client application to retrieve software version.

Parameters used are:

timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, the function will return with empty string. If timeout is 0, function will wait for client response until client is responding for max. time 3 seconds.

Return values are:

PChar string, that contains version of PG4UW control program, for example 3.07. If the function fails, return result is empty string ““.

function **SEND_CMD_CreditBox_GetTotalAvailableCredits**: integer; stdcall;

Function sends command “get total credits available” to PG4UW client application to retrieve available credits from all currently detected USB connected Credit boxes.

Parameters used are:

Function has not parameters.

Return values are:

0 or more - amount of available credits at found Credit box(es) connected
-1 - no connected Credit box(es) detected
-2 - unknown error

2.3 Constants used by remote control communication

Following constants define command codes and status codes used by remote communication in the direction Client -> Server. These constants can be tested by Server application in receive message Event handle procedure by which Server can recognize which commands Client sent to Server.
For practical use of the constants see the example Server application Pascal source code.

Constants are defined in unit RemoteConst.pas

```
//-------------------------------------
REMOTE_LIBRARY = 'remotelb.dll';

//----- commands Client -> Server  -----
TCP_KEYWORD_OPTYPE = 'optype';
TCP_KEYWORD_OPBUSY = 'opbusy';
TCP_KEYWORD_PROGRESS = 'progress';
TCP_KEYWORD_OPRESULT = 'opresult';
// codes for brief operation results
TCP_CMD_OPRESULT       = 'opresult:';
TCP_KEYWORD_OPRESULT_GOOD  = 'oprGood';
TCP_KEYWORD_OPRESULT_FAIL  = 'oprFail';
TCP_KEYWORD_OPRESULT_HWERR = 'oprHWError';
TCP_KEYWORD_OPRESULT_NONE  = 'oprNone';

// codes for detailed operation results
TCP_CMD_DETAILED_OPRESULT = 'detailedopresult:';
// value codes for detailed operation result
TDetailedOpResultValues = (  dorUnavailable = 0,
                             dorGood,
                             dorAdapterTestFailure,
                             dorInsertionTestFailure,
                             dorIDCheckFailure,
                             dorAccessCanceledByUserFailure,
                             dorOtherFailure,
                             dorOperationalFailure = 999 );

// codes for Load file/project result
TCP_CMD_LOAD_FILE_PRJ_RESULT   = 'loadresult:';
TCP_FILE_LOAD_GOOD             = 'frgood';
TCP_FILE_LOAD_ERROR            = 'frerror';
TCP_FILE_LOAD_CANCELLED        = 'frcancelled';

// codes for Select device result
TCP_CMD_SELECT_DEVICE_RESULT   = 'selectdeviceresult:';
TCP_SELECT_DEVICE_GOOD         = 'good';
TCP_SELECT_DEVICE_ERROR        = 'error';

// codes for Auto Select of EPROM/FLASH device
TCP_CMD_AUTSEL_EPRFLSH_RESULT  = 'autoseldeviceresult:'; // not used yet

// codes for server to client 'ready' question
TCP_CMD_CLIENT_READY_ANSWER = 'clienprogramisreadyanswer';
TCP_KEY_CLIENT_READY_YES    = 'isready';
TCP_KEY_CLIENT_READY_NO     = 'isnotready';

// codes for command line params result
TCP_CMD_PROCESS_CMDL_RESULT    = 'cmdlineparamsresult:';
TCP_CMD_PROCESS_CMDL_GOOD      = 'good';
TCP_CMD_PROCESS_CMDL_ERROR     = 'error';
```

3. Short example of remote control implementation

There are two basic ways how to use remote control:
- synchronous mode
- asynchronous mode

3.1 Synchronous mode

In synchronous mode when Server is sending any command to Client, Server will wait in cycle for Client response.
The Server jumps inside cycle until:
1. Server receives wished message from Client or
2. Server receives 'user break' request

3.2 Asynchronous mode

In asynchronous mode when Server is sending any command to Client, Server will not wait in cycle for Client response. The Server waits for Client messages indirectly by using Events. Procedure which handles messages from Client must be defined in parameter vProcessProc when creating Server communication object by calling procedure CreateServerAndMakeListenToClients.

Events are used both in synchronous and asynchronous modes. Procedure which handles messages from Client must be defined in parameter vProcessProc when creating Server communication object by calling procedure CreateServerAndMakeListenToClients.

To create remote control Server application, remote control functions have follow order as described bellow:

1. create Server object for remote TCP communication and start listening

```
CreateServerAndMakeListenToClients( ServerProcessProc,
                                    WriteToLogwindowProc,
                                    onClientConnectProc,
                                    onClientdisconnectProc,
                                    PChar(remote_ctrl_settings.Client_Server_Port));
```

2. run PG4UW Client application

3. when Server receives message that client is connected, Server can send commands SEND_CMD_...

It is recommended to ask Client if it is ready by **SEND_CMD_Quest_is_Client_Ready**.

If Client responds answer TCP_KEY_CLIENT_READY_YES it means Client is ready to receive executive commands (for example Load project, Program device and so on).

4. when closing Server application, call the **MakeClientServertCloseConnectionAndFree** procedure

4. Remote control examples - source files

Remote control examples are available in the subdirectory *\remotectrl\programs* placed in the directory where PG4UW control program is installed.

The typical path to remote control examples looks like this:

c:\Program Files\Manufacturer_sw\Programmer\remotectrl\programs

The Start Menu link to the examples is created also during installation of PG4UW,
if the “Create Start Menu shortcut” option is checked during installation.

Remote control examples are in the form of source files written in Pascal and C++ languages.

Source files are placed in the following subdirectories.

4.1 Source files written in Borland Delphi Pascal

PG4UWcmd - demonstration of remote control usage from Windows console application and command line

PG4UWrem - remote control usage from remote Windows GUI application, demonstrates basic remote control commands

RemoDemo - remote control usage from remote Windows application demonstrates implementation of Handler operator for automation of device programming

MultiDemo - demonstration of multiprogramming remote control, more PG4UW applications – more programmers – running on one computer at the same time

4.2 Source files written C++ (suitable for Borland C++ Builder and Microsoft Visual C++)

PG4UWcmd.vc - demonstration of remote control usage from Windows console application and command line

RemoDemo.bc - remote control usage from remote Windows application demonstrates implementation of Handler operator for automation of device programming (sources written for Borland C++ Builder only, because forms are used here)

4.3 Source files written in Microsoft Visual Basic 6

PG4UWcmd.VB6 - demonstration of remote control usage from Windows console application and command line

RemoDemo.VB6 - remote control usage from remote Windows application demonstrates implementation of Handler operator for automation of device programming

4.4 Source files written in Microsoft Visual Basic 2002/2003/2005 .NET

PG4UWcmd.VB.NET - demonstration of remote control usage from Windows console application and command line

RemoDemo.VB.NET - remote control usage from remote Windows application demonstrates implementation of Handler operator for automation of device programming

MultiDemo.VB.NET - demonstration of multiprogramming remote control, more PG4UW applications - more programmers - running on one computer at the same time

5. Using remote control with multiply programmers (multiprogramming)

Library **remotelb.dll** contains also set of functions which can be used for remote control of multiply programmers at the same time. Functions are very similar to “normal” functions, except that they have one additional parameter – index of Client (programmer), that has to be addressed by the remote command send from server (remote control Master) application.

Restrictions:

- multiprogramming feature is available for programmers (programming sites) connected via USB port only, therefore multiprogramming remote control can not be used for programmers, connected to PC via LPT port.
- remote control for multiprogramming (BeeHive4+, BeeHive204, N x BeeProg+, N x BeeProg2/ BeeProg2C, BeeProg2AP) is available for ISP programming only (for example implementation of programmers into ATE machines). For automated off line multiprogramming (programming in ZIF socket) customer can use BeeHive204AP/BeeHive204AP-AU, BeeHive304, N x BeeProg2AP and N x BeeProg3, programmer(s). Remote control of these programmers is done by remote control of PG4UWMC software. Ask us for details.
- the PG4UW remote control for multiprogramming is available only for registered programmers. Registration have to be done within 60 days after purchase of programmer. See please "Support » Product registration" section at our web site.
- let me note, the PG4UW remote control for multiprogramming is optional part of the software and is intentionally NOT PROMOTED part of programmers and software, therefore this feature could not be a reason for purchase of programmer(s). Therefore instead of complaining of above restrictions, please do not use this feature of software.
- remotelb.dll is not thread safe, therefore if you write multi-thread application, use please critical section.

Multiprogramming uses individual PG4UW.exe applications running instances for each of programmer used. To allow running more than one instance of PG4UW.exe at the same time, special command line parameters have to be specified, when starting each of PG4UW.exe instance.

Command line parameter syntax is following:

PG4UW.exe /usb:<index>:<programmer serial number>

index - specifies unique index, that will be used with the instance of PG4UW. The index allows to distinguish each instance of PG4UW when sending remote control commands from remote control software. Index is defined as number in range 1 to 16. The index value can also be considered to be Site number.

programmer serial number - specifies serial number of programmer, which has to be used with the instance of PG4UW application. The programmer with specified serial number will be searched on USB ports by the instance of PG4UW. There is allowed to use one programmer serial number in one PG4UW instance only at the same time.

Note:
In case that you are using our multiprogramming system, 'programmer serial number' mean 'site serial number'. To obtain site serial number please do following steps:
- run PG4UWMC
- press 'Search' button in main menu
- in dialog 'Search for Programmers' select proper programmer type and press 'Search' button
- in 'Programmers activity log' you can see site serial numbers for each site.

When connecting sites in your multiprogramming system, please start PG4UW instances sequentially. There exists a message, which is sent by instance successfully started and initialized, which means, that you can proceed with starting another instance.

Example:

We want to use multiprogramming with two programmers that have serial numbers 192-00123 and 192-00124. The procedure to run two instances of PG4UW is following:

1. the first instance for programmer with s/n: 192-00123 - in command line we specify:

PG4UW.exe /usb:1:192-00123

2. Then we should wait for response from started site

Answer from client:

- client sends response
TCP_CMD_SERVER_CAN_RUN_ANOTHER_SITE = 'canrunanothersite';

3. the second instance for programmer with s/n: 192-00124 can be started - in command line we specify:

PG4UW.exe /usb:2:192-00124

More on starting multiple sites can be found in example **MultiDemo** mentioned bellow.

When we want to use multiprogramming remote control, following command line parameters are useful to automatically enable remote control communication in PG4UW:

1. PG4UW.exe /usb:1:192-00123 /enableremote:autoanswer
2. PG4UW.exe /usb:2:192-00124 /enableremote:autoanswer

Remotelb.dll remote control with multiprogramming published functions

Following description presents purpose of each remote control function in multiprogramming mode. For more particular description of function declarations and parameters, please see the file *remotemulticlient.pas*.
If the remote Server application is written in C language, there is necessary to write appropriate header .h file to make functions from library *remotelb.dll* available in C/C++ project.
The Pascal unit file *remotemulticlient.pas* shows function declarations and parameters. This can be used for writing C/C++ header file.

5.1 General functions for Client/Server remote control connection establishing and connection parameters setting

To see the usage of following functions and procedures, the example of multiprogramming remote control application **MultiDemo** is available. The example is placed in the directory where PG4UW program is installed in the subdirectory *<PG4UW_inst_dir>\remotectrl\programs\MultiDemo*.
The complete path can look like this:
C:\Program Files\Manufacturer_sw\Programmer\remotectrl\programs\MultiDemo

```
procedure CreateServerAndMakeListenToMultiClients( MultiClientMode: BOOL;
                                                    vProcessProc: TProcWithPChar;
                                                    vWriteToLogProc: TProcWithPChar;
                                                    onClientConnectProc: TProc;
                                                    onClientDisconnectProc: TProc;
                                                    vPort: PChar); stdcall;
```

Procedure creates Server communication object, with defined parameters and starts waiting for Client(s). This procedure is used by Server application - remote control program.

Input parameters are:

MultiClientMode: BOOL - this parameter should be always true for multiprogramming mode.

vProcessProc: TProcWithPChar - define pointer to optional callback procedure which will be called every time Server receives message(s) from (PG4UW) Clients. This is usual way to receive information or commands from remote PG4UW Client.
If 'vProcessProc' is defined as **nil**, no “on-client command received“ event will be used. Then the only way how to receive information and commands from Client is reading of commands explicitly by calling of special function 'GetCommandStringFromFIFO_CINDEX'. For more details please take a look at description of function GetCommandStringFromFIFO_CINDEX.

vWriteToLogProc: TProcWithPChar - define pointer to optional callback procedure which is useful especially during debugging of Server program. Procedure is called when any of basic Client-Server communication events occures. Communication events are: connect/disconnect Client-Server, send message to Client, receive message from Client. Procedure can contain user defined write to memo or log window of Server application. If it is defined as **nil**, no “write-to-log“ events will be used.

onClientConnectProc: TProc - optional callback procedure is called as event when Client is connected to Server. If it is defined as **nil**, no “on-client connect“ event will be used.

onClientDisconnectProc: TProc - optional callback procedure is called as event when Client is disconnected from Server. If it is defined as **nil**, no “on-client disconnect“ event will be used.

vPort: PChar - defines port for TCP communication (default is 'telnet')

Server does not have address defined itself. Internally Server has defined address 0.0.0.0, which means, that Server accepts Clients from all interfaces.

function **GetIfClientAppWithOrdNumExists**(index: integer): BOOL; stdcall;

Function is used by Server application. Function returns true, if Client with “index” number is connected to Server, otherwise returns false.

function **GetIfAnyClientAppIsConnectedToServer**: BOOL; stdcall;

Function is used by Server application. Function returns true, if at least one Client is connected to Server, otherwise returns false.

5.2 Procedures used for sending basic types of commands from remote Server to PG4UW (Client)

procedure **SEND_CMD_BringToFront_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'bringtofront' to Client with index specified by parameter “index”. When Client receives the message, it tries to make BringToFront operation. BringToFront operation is basically activation of Client application window.

Example:

Server request:

SEND_CMD_BringToFront_CINDEX;

Answer from client:

- client send no answer

procedure **SEND_CMD_ShowMainForm_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'showmainform' to Client with index specified by parameter

"index". When Client receives the message, it makes Show command for main window of the Client application. The message does nothing if the main window of Client application is already visible. Please use this command only in situations when no operation is currently running in the Client application.

Example:

Server request:

SEND_CMD_ShowMainForm_CINDEX(1);

Answer from client:

- client send no answer

procedure **SEND_CMD_HideMainForm_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'hidemainform' to Client with index specified by parameter "index". When Client receives the message, it makes Hide command for main window of the Client application. Also Client's icon placed on taskbar will be hidden and small icon appears on the tray panel. The message does nothing if the main window of Client application is already hidden. Please use this command only in situations when no operation is currently running in the Client application.

Example:

Server request:

SEND_CMD_HideMainForm_CINDEX(1);

Answer from client:

- client send no answer

procedure **SEND_CMD_BlankCheckDevice_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'blankcheck' to Client with index specified by parameter "index". When Client receives the message, it starts device Blank check operation. In the end of operation Client sends operation result to Server. Operation result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.
Client PG4UW can accept and start 'blankcheck' command only when no operation is currently running in PG4UW. If PG4UW is running some operation, the 'blankcheck' command will be ignored. To receive the current status of programmer, use command SEND_CMD_GetProgStatus described later.

Example 1:

Server request:

SEND_CMD_BlankCheckDevice_CINDEX(1);

Answer from client:

See the Client send operation result chapter.

Example 2:

```
procedure SiteStartDeviceProgramming(siteindex: integer);
begin
        // check index range
        if (siteindex < 1) or (siteindex > MAX_SITE_COUNT) then exit;
```

```
        // check if Site is Ready and not busy
        if CheckIfTheSiteIsBusy(siteindex) = true then
        begin
                ShowMessage(Format('Sorry, the Site #%d is busy now. Operation can not be started immediately.'#13#10 +
                'Please try again later.', [siteindex]));
                exit;
        end; { if CheckIfTheSiteIsBusy(siteindex) = true... }

        // preventively send command "Stop", only when current operation detected
        // on Site is different from operation 'Blank check device'
        if TAppOperationType(device_progress[siteindex].opcode) <> otblankCheck then
        begin
                 SEND_CMD_Stop_CINDEX(siteindex);
                 application.processmessages;
                 sleep(250);
        end; { if }

        { there can be send one of following commands:
        SEND_CMD_BringToFront_CINDEX(index);
        SEND_CMD_BlankCheckDevice_CINDEX(index);
        SEND_CMD_ReadDevice_CINDEX(index);
        SEND_CMD_VerifyDevice_CINDEX(index);
        SEND_CMD_ProgramDevice_CINDEX(index);
        SEND_CMD_EraseDevice_CINDEX(index);
        SEND_CMD_Stop_CINDEX(index);
        SEND_CMD_LoadProject_CINDEX(index, PChar(PRJstr));
        }

        // send command "Program device" to Site client PG4UW
        SEND_CMD_BlankCheckDevice_CINDEX(index);

        application.processmessages;
        sleep(0);
end;
```

procedure **SEND_CMD_VerifyDevice_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'verifydevice' to Client with index specified by parameter “index”. When Client receives the message, it starts device Verify operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

```
SEND_CMD_VerifyDevice_CINDEX(1);
```

Answer from client:

See the Client send operation result chapter.

procedure **SEND_CMD_ProgramDevice_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'programdevice' to Client with index specified by parameter “index”. When Client receives the message, it starts device Program operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

```
SEND_CMD_ProgramDevice_CINDEX(1);
```

Answer from client:

See the Client send operation result chapter.

procedure **SEND_CMD_EraseDevice_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'erasedevice' to Client with index specified by parameter “index”. When Client receives the message, it starts device Erase operation. In the end of operation Client sends operation result to Server. Other properties of PG4UW behaviour are same as for command procedure SEND_CMD_BlankCheckDevice.

Example:

Server request:

SEND_CMD_EraseDevice_CINDEX(1);

Answer from client:

See the Client send operation result chapter.

Description of **Client send operation result** for device operations

There are available two commands for device operation result status. Both commands are sent automatically from PG4UW to remote control application after device operation was finished. It is recommended, the remote control application software uses only one of them at once.

1. command ***TCP_CMD_OPRESULT*** = *'opresult';*

   The command offers brief information about device operation result. It contains one of following return codes:

```
// code for brief operation result
TCP_CMD_OPRESULT          = 'opresult:';
// value codes for operation result
TCP_KEYWORD_OPRESULT_GOOD  = 'oprGood';
TCP_KEYWORD_OPRESULT_FAIL  = 'oprFail';
TCP_KEYWORD_OPRESULT_HWERR = 'oprHWError';
TCP_KEYWORD_OPRESULT_NONE  = 'oprNone';

// answer will be in one of following string form
TCP_CMD_OPRESULT + TCP_KEYWORD_OPRESULT_GOOD
// or
TCP_CMD_OPRESULT + TCP_KEYWORD_OPRESULT_FAIL
// or
TCP_CMD_OPRESULT + TCP_KEYWORD_OPRESULT_HWERR
to uz je // or
TCP_CMD_OPRESULT + TCP_KEYWORD_OPRESULT_NONE
```

2. command ***TCP_CMD_DETAILED_OPRESULT*** = *'detailedopresult:';*

   The command offers more detailed information about device operation result. It contains one of following return codes:

```
// code for detailed operation result
TCP_CMD_DETAILED_OPRESULT = 'detailedopresult:';
// value codes for detailed operation result
TDetailedOpResultValues = ( dorUnavailable = 0,
                            dorGood,
                            dorAdapterTestFailure,
                            dorInsertionTestFailure,
                            dorIDCheckFailure,
                            dorAccessCanceledByUserFailure,
                            dorOtherFailure,
                            dorOperationalFailure = 999 );
```

   Example for answer string of detailed operation result received from PG4UW:

```
// example in Delphi Pascal language
var
  OpResultValue: TDetailedOpResultValues;
  ResultStr: string;
...
ResultStr := Format("%s%d", [TCP_CMD_DETAILED_OPRESULT, Integer(OpResultValue)]);
```

procedure **SEND_CMD_RepeatLastDevOperation_CINDEX**(index: integer); stdcall;

Procedure is used to send lastly used device operation command to Client with index specified by parameter “index”. For example if lastly used command is SEND_CMD_ProgramDevice, the call of procedure SEND_CMD_RepeatLastDevOperation will be the same as call of SEND_CMD_ProgramDevice.

procedure **SEND_CMD_Stop_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'stopoperation' to Client with index specified by parameter “index”. When Client receives the message, it stops current device operation. If no operation is running, the 'stopoperation' command does nothing. Other function of 'stopoperation' is closing message window(s) in Client application.

Example:

Server request:

SEND_CMD_Stop_CINDEX(1);

Answer from client:

- client send no answer

procedure **SEND_CMD_CloseApp_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'closeapp' to Client with index specified by parameter “index”. When Client receives the message, it makes Close command for main window of the Clients application. The Close command can be properly performed when no operation is currently running in the Client application. Please use the SEND_CMD_CloseApp command when no operation on device is running and no modal dialogs are shown. To ensure that no operation is currently running, command **SEND_CMD_GetProgStatus** can be used. Also commands **SEND_CMD_Stop** for operation stopping can be used.
For more details see enclosed examples of remote control written in Pascal and C++ languages. Examples are described in the chapter III. of this manual.

Example 1:

Server request:

SEND_CMD_CloseApp_CINDEX(1);

Answer from client:

- client send no answer

Example 2:

```
{ Procedure sends remote command SEND_CMD_CloseApp_CINDEX to Site
with index siteindex. Procedure also checks the Site is not busy.
If the Site is busy (currently running device operation), the procedure
shows warning message and does not perform the close command. }
procedure SiteCloseSiteApp(siteindex: integer);
const
        MAX_TIMEOUT_FOR_DISCONNECT = 8000{ms};
var
        start_time: DWORD;
begin
        // check index range
```

```
        if (siteindex < 1) or (siteindex > MAX_SITE_COUNT) then exit;

        // check if Site is Ready and not busy
        if CheckIfTheSiteIsBusy(siteindex) = true then
        begin
                ShowMessage(Format('Sorry, the Site #%d is busy now. Operation can not be started immediately.'#13#10 +
                'Please try again later.', [siteindex]));
                exit;
        end; { if CheckIfTheSiteIsBusy(siteindex) = true... }

        if GetIfClientAppWithOrdNumExists(siteindex) = true then
        try
                // write info about disconnecting of Sites to Log window
                WriteToLogwindowProc('Disconnecting Sites, wait please...');

                SEND_CMD_CloseApp_CINDEX(siteindex);
                application.processmessages;

                // wait for disconnecting of Site
                start_time := GetTickCount;
                while GetIfClientAppWithOrdNumExists(siteindex) = true do begin
                        sleep(100);
                        application.processmessages;
                        // check also timeout
                        if (GetTickCount - start_time) > MAX_TIMEOUT_FOR_DISCONNECT then break;
                end; { while }
        finally
                // write info about Sites disconnection completion to Log window
                WriteToLogwindowProc('Sites successfuly disconnected.');
        end; { try }
end;
```

procedure **SEND_CMD_SelectDevice_CINDEX**(index: integer; devmanuf, devname: PChar); stdcall;

Procedure is used to send message 'selectdevice:' to Client with index specified by parameter “index”. When Client receives the message, it tries to select specified device. Device is specified by parameters devmanuf and devname. Parameters are not case sensitive. The Client sends select device result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

Example: To select device Intel TE28F160C3B [TSOP48] call

Server request:

```
SEND_CMD_SelectDevice_CINDEX(1, 'Intel', 'TE28F160C3B [TSOP48]');
```

Answer from client:

- client send operation result

```
// codes for operations result
TCP_CMD_SELECT_DEVICE_RESULT   = 'selectdeviceresult:';
TCP_SELECT_DEVICE_GOOD         = 'good';
TCP_SELECT_DEVICE_ERROR        = 'error';

//answer will be in followed form
TCP_CMD_SELECT_DEVICE_RESULT + TCP_SELECT_DEVICE_GOOD
//or
TCP_CMD_SELECT_DEVICE_RESULT + TCP_SELECT_DEVICE_ERROR
```

procedure **SEND_CMD_EPROMFLASH_AutoSelect_CINDEX**(index: integer;
pinsnumber: PChar); stdcall;

Procedure is used to send message 'autoseldevice:' to Client with index specified by parameter “index”. When Client receives the message, it starts autoselect device operation. Parameter

pinsnumber can be used to specify the pin number of device which helps more reliable detection of device type. If parameter pinsnumber is blank Pchar (''), autoselect operation tries to detect inserted device pin number automatically. The Client sends currently selected device to Server but not result of autoselect detection success.

Example:

Server request:

```
SEND_CMD_EPROMFLASH_AutoSelect_CINDEX(1, ''); //device pins detected automatically

SEND_CMD_EPROMFLASH_AutoSelect_CINDEX(1, '48'); //48-pins device
```

Answer from client:

```
- client send currently selected device

// codes for operations result
TCP_CMD_AUTSEL_EPRFLSH_RESULT  = 'autoseldeviceresult:';

//answer will be in followed form
TCP_CMD_AUTSEL_EPRFLSH_RESULT + 'manufacturer' + '  ' + 'device name' + ' '
```

procedure **SEND_CMD_LoadProject_CINDEX**(index: integer; prjname: PChar); stdcall;

Procedure is used to send message 'loadproject:' to Client with index specified by parameter “index”. When Client receives the message, it tries to load project file specified by parameter prjname. The Client sends load project result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

Example:

Server request:

```
SEND_CMD_LoadProject_CINDEX(1, "C:\test.eprj');
```

Answer from client:

```
- client send project load result

// codes for Load file/project result
TCP_CMD_LOAD_FILE_PRJ_RESULT   = 'loadresult:';
TCP_FILE_LOAD_GOOD             = 'frgood';
TCP_FILE_LOAD_ERROR            = 'frerror';

//answer will be in followed form
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_GOOD
//or
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_ERROR
```

procedure **SEND_CMD_GetProjectFileChecksum_CINDEX**(index: integer; prjname: PChar); stdcall;

Procedure is used to send message 'getprojectfilechecksum:' to Client with index specified by parameter “index”. When Client receives the message, it tries to compute CRC-32 of project file specified by parameter prjname. The Client sends result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

procedure **SEND_CMD_LoadFile_CINDEX**(index: integer; fname: PChar); stdcall;

Procedure is used to send message 'loadfile:' to Client with index specified by parameter “index”.

When Client receives the message, it tries to load file specified by parameter fname. The Client sends load file result to Server. The result command received from client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.

Example:

Server request:

```
SEND_CMD_LoadFile_CINDEX(1, "C:\test.bin');
```

Answer from client:

```
- client send file load result

// codes for Load file/project result
TCP_CMD_LOAD_FILE_PRJ_RESULT   = 'loadresult:';
TCP_FILE_LOAD_GOOD             = 'frgood';
TCP_FILE_LOAD_ERROR            = 'frerror';
TCP_FILE_LOAD_CANCELLED        = 'frcancelled';

//answer will be in followed form
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_GOOD
//or
TCP_CMD_LOAD_FILE_PRJ_RESULT + TCP_FILE_LOAD_ERROR
//or
TCP_CMD_LOAD_FILE_PRJ_RESULT +  TCP_FILE_LOAD_CANCELLED
```

procedure **SEND_CMD_GetProgStatus_CINDEX**(index: integer); stdcall;

Procedure is used to send message 'getprogstatus' to Client with index specified by parameter “index”. When Client receives the message, it sends its current status info to Server. The status info command received from Client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.
Status info contains four basic info status data:
- busy status
- current device operation type
- current device operation progress

For more information see the example unit

Example 1:

Server request:

```
SEND_CMD_GetProgStatus_CINDEX(1);
```

Answer from client:

```
- client current status information

//operation_code enumeration type
operation_code =(otNone, otRunApp, otblankCheck, otReadDevice, otVerifyDevice,
                 otProgramDevice, otEraseDevice, otPlayPlayer, otLoadProject, otStop,
                 otSelfTest, otSelfTestPlus);

//codes for current status information
TCP_KEYWORD_OPTYPE = 'optype';
TCP_KEYWORD_PROGRESS  = 'progress';
TCP_CMD_PROGRAMMER_READY_STATUS = 'programmerreadystatus:';
TCP_KEY_PROGRAMMER_NOTFOUND = 'notfound';
TCP_KEY_PROGRAMMER_READY    = 'ready';

TCP_KEYWORD_OPTYPE + ':' + operation_code + ' '  +
TCP_KEYWORD_PROGRESS +' :'  + device_progress + ' '  +
TCP_CMD_PROGRAMMER_READY_STATUS + TCP_KEY_PROGRAMMER_READY
//or
```

```
TCP_KEYWORD_OPTYPE + ':' + operation_code + ' ' +
TCP_KEYWORD_PROGRESS +' :' + device_progress + ' ' +
TCP_CMD_PROGRAMMER_READY_STATUS + TCP_KEY_PROGRAMMER_NOTFOUND
```

Example 2:

send request to client:

```
SEND_CMD_GetProgStatus_CINDEX(1);
```

processing the response from client:

```
procedure Process_KEYWORD_OPTYPE(index: integer; CMDline: ansistring);
var
        tmpstr: string;
        progress, errcode, q: integer;
begin
        // find out current operation type
        q := 1; //pos of TCP_KEYWORD_OPTYPE
        tmpstr := copy(CMDline, q + length(TCP_KEYWORD_OPTYPE)+1, length(CMDline));
        q := pos(' ', tmpstr);
        if q > 0 then tmpstr := trim(copy(tmpstr, 1, q-1))
        else tmpstr := 'unknown';

        // set cur op type
        val(tmpstr, q, errcode);
        if errcode = 0 then device_progress[index].opcode := q
        else device_progress[index].opcode := 0;

        // find out current operation progress
        q := pos(TCP_KEYWORD_PROGRESS, CMDline);
        if q > 0 then
        begin
                tmpstr := copy(CMDline, q + length(TCP_KEYWORD_PROGRESS)+1, length(CMDline));
                q := pos(' ', tmpstr);
                if q > 0 then tmpstr := trim(copy(tmpstr, 1, q-1))
                else tmpstr := 'unknown';

                val(tmpstr, progress, errcode);
                if errcode = 0 then device_progress[index].progress := progress
                else device_progress[index].progress := 0;
        end; { if q > 0... }

        // find out current programmer status - Ready or Not found (Demo mode)
        q := pos(TCP_CMD_PROGRAMMER_READY_STATUS, CMDline);
        if q > 0 then
        begin
                tmpstr := copy(CMDline, q + length(TCP_CMD_PROGRAMMER_READY_STATUS), length(CMDline));
                q := pos(' ', tmpstr);
                if q > 0 then tmpstr := trim(copy(tmpstr, 1, q-1))
                else tmpstr := trim(tmpstr);
                if tmpstr = TCP_KEY_PROGRAMMER_READY then
                        device_progress[index].pgm_status := true   // programmer is Ready
                else
                        device_progress[index].pgm_status := false; // programmer is in Demo mode
        end; { if q > 0... }
        // refresh status indicators
        if formMultiPanel <> nil then
                if device_progress[index].opcode = integer(otnone) then
                        with formMultiPanel do begin
                                Set_progressbar_value(index,
                                device_progress[index].progress,
                                device_progress[index].busy);
                                // reset result flag if no operation is active
                                device_result[index] := oprNone;
                                formMultiPanel.SetLEDcolors(index,
                                device_progress[index].busy,
                                device_result[index]);
                        end; { with }
end; { procedure }
```

procedure **SEND_CMD_Quest_is_Client_Ready_CINDEX**(index: integer); stdcall;

Procedure sends question to Client, if the Client is Ready. Client sends ready yes/no status message to Server. The ready status command received from Client can be processed by Server application in procedure defined by pointer parameter vProcessProc in procedure CreateServerAndMakeListenToClients.
Client is ready when no device operation is currently running and main Client's window is not hidden by any modal dialogs in Client.

Example:

Request from server:

```
SEND_CMD_Quest_is_Client_Ready_CINDEX(1);
```

Answer from client:

```
// codes for server to client 'ready' question
TCP_CMD_CLIENT_READY_ANSWER = 'clienprogramisreadyanswer';
TCP_KEY_CLIENT_READY_YES    = 'isready';
TCP_KEY_CLIENT_READY_NO     = 'isnotready';

//answer will be in followed form
TCP_CMD_CLIENT_READY_ANSWER + TCP_KEY_CLIENT_READY_YES;
//or
TCP_CMD_CLIENT_READY_ANSWER + TCP_KEY_CLIENT_READY_NO;
```

function **SEND_CMD_GetProgrammerConnectedStatus_CINDEX**(index: integer); stdcall;

Function is suitable to detect, if the programmer Site with index “index” is successfully connected and present in PG4UW application associated with the Site. Function sends command "get programmer connection status" to client PG4UW application and waits for client response (timeout is 3 seconds).

Function has no parameters.
Return values are:
True - OK - programmers is successfully connected
False - BAD - programmer is not found or no response received (timeout occurs)

function **GetCommandStringFromFIFO_CINDEX**(index: integer): PChar; stdcall;

Function is used to read next remote received information or control command from Client (PG4UW) application with index specified by parameter 'index'. The commands received from client are stored internally in remotelb.dll in FIFO buffer. User can access the FIFO sequentially by this function. Each call of this function will pick the oldest command from FIFO buffer and decrements internal FIFO counter by 1. If the number of items in FIFO is 0, function will return value *nil*.
Return values are:
If the function succeeds, the return value is pointer to command string read from FIFO.
If the function fails, the return value is *nil*.
Function fails, if it is called while FIFO buffer of commands is empty or FIFO does not exist.

This function is especially useful, when user does not use callback functions in server remote control application. In such case, the commands received from PG4UW Client can be read by using function GetCommandStringFromFIFO_CINDEX. More details about using the function, can be seen in remote control examples.

Notes:
1. The function is active, only when callback procedure pointer specified by parameter 'vProcessProc' in remote control initialization procedure "CreateServerAndMakeListenToMultiClients" is nil. If there is specified callback procedure (vProcessProc <> nil), the function GetCommandStringFromFIFO is

not usable, and it will always return value nil.

2. The FIFO buffer has size 128 items. It means, it can store maximally of 128 commands received from Client PG4UW. So it is recommended to call function GetCommandStringFromFIFO_CINDEX periodically to pop commands from FIFO and prevent FIFO buffer from getting full. If the number of commands received is greater than capacity of FIFO memory, the newly coming commands are ignored. Of course the Client does not send commands or information to Server application without requiring them from Server. So reading commands from FIFO has sense in cases, the Server sends commands to Client.

3. Commands received by function 'GetCommandStringFromFIFO_CINDEX' are in form of null-terminated strings. The type of null-terminated string is known as:
   - PChar in Pascal language
   - char * in C/C++

   A null-terminated string is a zero-based array of characters that ends with NULL (#0); since the array has no length indicator, the first NULL character marks the end of the string.

function **SEND_CMD_WBUFFER_CINDEX**(index: integer;
bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to write specified block of data to main device buffer of remote client (PG4UW program).
Parameters used are:
index - index of Client (programmer Site)
bufaddr - buffer address in PG4UW, where data have to be written
bufptr - pointer to memory block of data, which have to be sent to PG4UW
bufsize - number of bytes, which have to be sent to PG4UW, valid values are from 1 to 256 bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_WBUFFER_CINDEX will return error code. If TimeOut is 0, function SEND_CMD_WBUFFER_CINDEX will wait for client response until client is responding for max. time 3 seconds.
Return values are:
0 - OK - data was sent successfuly
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_RBUFFER_CINDEX**(index: integer;
bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to read specified region of main device buffer from PG4UW remote client to memory region specified by pointer parameter bufptr.
Parameters used are:
index - index of Client (programmer Site)
bufaddr - buffer start address in PG4UW, data are read from this address
bufptr - pointer to memory block, where data has to be copied
bufsize - number of bytes, which have to be read from PG4UW buffer, valid values are from 1 to 256

bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_RBUFFER_CINDEX will return error code. If TimeOut is 0, function SEND_CMD_RBUFFER_CINDEX will wait for client response until client is responding for max. time 3 seconds.

Return values are:
0 - OK - data was sent successfully
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occurred, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_WBUFFER_EX_CINDEX**(index: integer;
buffindex: integer;
bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to write specified block of data to main device buffer of remote client (PG4UW program). The function is very similar to function **SEND_CMD_WBUFFER_CINDEX**. The only difference is one more parameter – buffindex, which is used to specify buffer to be used for writing data.

Parameters used are:
index - index of Client (programmer Site)
buffindex - specifies the order of buffer, where data will sent, the main buffer has index '1'. The first secondary buffer has index '2', etc. Please note, the secondary buffer(s) is(are) available for some kinds of devices only (e.g. Microchip PIC16F628). The kind of buffer indexed by parameter buffindex depends on order of buffer in application PG4UW in dialog View/Edit buffer. For example device Microchip PIC16F628 has additional buffer with label "Data EEPROM". This buffer can be accessed for data write(s) by this function when buffindex = 2 is specified.
bufaddr - buffer address in PG4UW, where data have to be written
bufptr - pointer to memory block of data, which have to be sent to PG4UW
bufsize - number of bytes, which have to be sent to PG4UW, valid values are from 1 to 256 bytes
timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_WBUFFER_EX_CINDEX will return error code. If TimeOut is 0, function SEND_CMD_WBUFFER_EX_CINDEX will wait for client response until client is responding for max. time 3 seconds.

Return values are:
0 - OK - data was sent successfuly
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_RBUFFER_EX_CINDEX**(index: integer;
buffindex: integer;
bufaddr: longint;
bufptr : pointer;
bufsize: integer;
timeout: integer): integer; stdcall;

Function is used to read specified region of main device buffer from PG4UW remote client to memory region specified by pointer parameter bufptr. The function is very similar to function **SEND_CMD_RBUFFER_CINDEX**. The only difference is one more parameter – buffindex, which is used to specify buffer to be used for reading data from.

Parameters used are:

index - index of Client (programmer Site)

buffindex - specifies the order of buffer, where data will sent, the main buffer has index '1'. The first secondary buffer has index '2', etc. Please note, the secondary buffer(s) is(are) available for some kinds of devices only (e.g. Microchip PIC16F628). The kind of buffer indexed by parameter buffindex depends on order of buffer in application PG4UW in dialog View/Edit buffer. For example device Microchip PIC16F628 has additional buffer with label "Data EEPROM". This buffer can be accessed for data write(s) by this function when buffindex = 2 is specified.

bufaddr - buffer start address in PG4UW, from data has to be read

bufptr - pointer to memory block of data, where data has to be copied

bufsize - number of bytes, which have to be read from PG4UW buffer, valid values are from 1 to 256 bytes

timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, function SEND_CMD_RBUFFER_EX_CINDEX will return error code. If TimeOut is 0, function SEND_CMD_RBUFFER_EX_CINDEX will wait for client response until client is responding for max. time 3 seconds.

Return values are:

0 - OK - data was sent successfuly
1 - address parameter bufaddr is out of range
2 - sent buffer data size parameter bufsize is out of range
3 - timeout occured, client is not responding
4 - TCP communication object does not exist or not connected
5 - unknown error

function **SEND_CMD_GET_PG4UW_VERSION_CINDEX**(index: integer; timeout: integer): PChar; stdcall;

Function sends command “get sw version” to PG4UW client application of Site #<index> to retrieve software version.

Parameters used are:

index - index of Client (programmer Site, index = 1, 2, …, n)

timeout - the timeout parameter specifies the amount of time (in milliseconds) to wait for a response from the remote client. If client is not answering during the time specified by timeout, the function will return with empty string. If timeout is 0, function will wait for client response until client is responding for max. time 3 seconds.

Return values are:

PChar string, that contains version of PG4UW control program, for example 3.07. If the function fails, return result is empty string ““.

function **SEND_CMD_CreditBox_GetTotalAvailableCredits_CINDEX**(index: integer): Integer;stdcall;

Function sends command “get total credits available” to PG4UW client application of Site #<index> to retrieve available credits from all currently detected USB connected Credit boxes.

Parameters used are:

index - index of Client (programmer Site, index = 1, 2, …, n)

Return values are:

0 or more - amount of available credits at found Credit box(es) connected
-1 - no connected Credit box(es) detected
-2 - unknown error

Note: For mutiprogramming systems, there is no matter, what Site is used to retrieve current status of available total credits. The credit status is common for all Sites connected to the computer.

6. Remote command line control of PG4UW

6.1 Configuration command line parameters

The remote control function of PG4UW application(s) needs the remote control feature enabled. This can be done by one of two ways:

1. enabling remote control by checking check box “Enable program remote control” in PG4UW's “General options” dialog. The dialog is accessible by menu command “Options | General options”. Remote control options are placed in that dialog at page “Remote control”.
2. enabling remote control by using command line parameter */enableremote[:autoanswer]*

It is recommended to use remote control command line parameter instead of manual remote control options setting.

The command line parameter has format

/enableremote[:autoanswer]

The optional part :autoanswer is required for multiprogramming remote control (controlling more than one instance of PG4UW at the same time) and is optional for single programming remote control.

The *autoanswer* option modifies behavior of PG4UW application as following:

1. when */enableremote:autoanswer* option is used:
   - when any warning/information/question/error message is displayed, PG4UW does not show the message normally, but only writes its content to Programmer activity log and simulates user answer by clicking on default button of the message
   - PG4UW makes *Repeat?* dialog remaining displayed after device operation is complete. This allows to see the device operation result in Info window and also in remote control application. To close *Repeat?* dialog from remote control, command **SEND_CMD_Stop** can be used. When just repeating of device operation is needed, there can be send remote command of current device operation. No **SEND_CMD_Stop** command is needed.
2. when */enableremote* only option is used (without *autoanswer* optional part):
   - when any warning/information/question/error message is displayed, PG4UW shows the message normally and waits for user confirmation (by click on message button)
   - PG4UW does not display *Repeat?* dialog after device operation is complete. It means, the PG4UW closes Info window displayed during device operation and stay in main PG4UW window focused.
3. when */enableremote:autoanswer* option is used in multiprogramming mode (together with parameter /usb:<sitenum>:<site_sn>):
   - PG4UW starts in hidden mode, displaying small green icon with site number #sitenum in Windows system tray.

   Tip: If you wish to start PG4UW application in visible state (not hidden), even if parameter /enableremote:autoanswer is used, use also additional command line parameter */startvisible*

6.2 Executive command line parameters

PG4UW accept set of commands from the command line (command line parameters). The remote control can be achieved also by this command line parameters, but more efficient way is to use special tool **PG4UWcmd.exe**, which has many advantages. The main advantage is size of the **PG4UWcmd**, which result the calling of **PG4UWcmd** results a much faster response than calling of PG4UW directly.

Program PG4UWcmd.exe can be used to:
1. start PG4UW application with specified command line parameters
2. force command line parameters to PG4UW that is already running

Very good feature of PG4UWcmd.exe is its return code according to command line parameters operation result in PG4UW.

6.3 Return values of PG4UWcmd.exe

If the command line parameters processed in PG4UW were successful, the Exitcode (or ErrorLevel) of PG4UWcmd.exe is zero. Otherwise the ExitCode value is number 1 or more.
Return value of program PG4UWcmd.exe can be tested in batch files.

Following executive command line parameters are available to use with PG4UWcmd.exe

/Prj:<file_name>

Loads project file. Parameter <file_name> means full or relative project file path and name.

/Loadfile:<file_name>

Loads file. Parameter <file_name> means full or relative path to file that has to be loaded. File format is detected automatically

/Program[:switch]

Forces start of "Program device" operation automatically when program is starting, or even if program is already running. Also one of following optional switches can be used:

1. switch 'noquest' forces start of device programming without question
2. switch 'noanyquest' forces start of device programming without question and after operation on device is completed, program doesn't show "Repeat" operation dialog and goes directly into main program window.

Examples:

1. /Program
2. /Program:noquest
3. /Program:noanyquest

/Saveproject:<file_name>

The command is used to save currently selected device type, buffer contents and configuration to project file. Command */Saveproject:...* is equivalent to user selected command ***Save project*** in PG4UW control program.

/Close

Parameter **/Close** has sense together with /Program parameter only, and makes program to close automatically after device programming is finished successfully.

/Close:always

Parameter **/Close:always** has sense together with /Program parameter only, and makes program to close automatically after device programming is finished, no matter if device operation was successful or not.

/Demo

Parameter **/Demo** is very useful when no programmer hardware is available, and user wants to remote control PG4UW software anyway. If the parameter is specified, PG4UW application will not make search for programmers connected. It automatically selects suitable programmer and sets its state to “Not connected”. Parameter /Demo can be used with other command line parameters, and we recommend to use file or buffer parameters only. Parameter **/Program[:switch]** will not work when /Demo parameter is specified.

/Eprom_Flash_Autoselect[:xx]

Forces automatic select EPROM or FLASH type by reading of electronic ID from the chip, inserted currently in ZIF socket of programmer. Optional parameter **xx** means pins number of device in ZIF (this time are valid 28 or 32 pins only) and it is required just for older programmers without insertion test capability. For others programmers the xx parameter can be omitted, because is ignored.

Examples:

/Eprom_Flash_Autoselect
/Eprom_Flash_Autoselect:32

/writebuffer:ADDR1:B11,B12,B13,B14,...,B1N[::ADDR2:B21,B22,B23,B24,...,B2M]...

Command /writebuffer is used to write block of Bytes to PG4UW main buffer at specified address. Write buffer command has one block of data required and other block(s) of data (marked with [...]) optional. Please do not use spaces or tabs in the command.
Buffer address is always defined as Byte address, it means, that for buffer organisation x16, the address AAAAx16 in buffer has to be specified in command /writebuffer as 2*AAAA (x8).

Example 1:

/writebuffer:7FF800:12,AB,C5,D4,7E,80

Writes 6 Bytes 12H ABH C5H D4H 7EH 80H to buffer at address 7FF800H.
The addressing looks like following:
the first Byte at the lowest address

Buffer – Address	*Data*
7FF800H	*12H*
7FF801H	*ABH*
7FF802H	*C5H*
7FF803H	*D4H*
7FF804H	*7EH*
7FF805H	*80H*

Example 2:

/writebuffer:7FF800:12,AB,C5,D4,7E,80::FF0000:AB,CD,EF,43,21
_____/ _____/
the first block of data the second block of data

Writes two blocks of data to buffer.

The first block of data - 6 Bytes 12H ABH C5H D4H 7EH 80H are written to buffer at address 7FF800H in the same way as in Example 1.

The second block of data - 5 Bytes ABH CDH EFH 43H 21H are written to buffer at address FF0000H.
The addressing looks like following:

the first Byte at the lowest address
Buffer – Address Data
FF0000H ABH
FF0001H CDH
FF0002H EFH
FF0003H 43H
FF0004H 21H

/writebufferex:INDEX:ADDR1:B11,B12,B13,B14,...,B1N[::ADDR2:B21,B22,B23,B24,...,B2M]...

Command /writebufferex is used to write block of Bytes to PG4UW main buffer at specified address. The command is very similar to command /writebuffer, except one more parameter – INDEX.

The INDEX parameter specifies the order of buffer, where data will sent. The main buffer has index '1'. The first secondary buffer has index '2', etc. Please note, the secondary buffer(s) is(are) available for some kinds of devices only (e.g. Microchip PIC16F628). The kind of buffer indexed by parameter buffindex depends on order of buffer in application PG4UW in dialog View/Edit buffer. For example device Microchip PIC16F628 has additional buffer with label "Data EEPROM". This buffer can be accessed for data write(s) by this function when buffindex = 2 is specified.

Example 1:

/writebufferex:1:7FF800:12,AB,C5,D4,7E,80

The command is equivalent to command
/writebuffer:1:7FF800:12,AB,C5,D4,7E,80
described in section about command /writebuffer.

Example 2:

/writebufferex:2:2F:12,AB,C5,D4,7E,80

The command writes 6 Bytes 12H ABH C5H D4H 7EH 80H to secondary buffer with index "2" at address 2FH. The addressing looks like following:
the first Byte at the lowest address
Buffer – Address Data
00002FH 12H
000030H ABH
000031H C5H
000032H D4H
000033H 7EH
000034H 80H

6.4 Basic rules for using of executive command line parameters

1. program PG4UWcmd.exe must be located in the same directory as program PG4UW.exe
2. if PG4UW.exe is not running when PG4UWcmd.exe is called, it will be automatically started
3. command line parameters are not case sensitive
4. command line parameters can be used when first starting of program or when program is already running
5. if program is already running, then any of command line operation is processed only when program was not busy (no operation was currently executing in program). Program must be in basic state, i.e. main program window focused, no modal dialogs displayed, no menu commands opened or executed
6. order of processing command line parameters when using more parameters together is defined firmly as following:

   step1 Load project (/Prj:...)
   step2 Load file (/Loadfile:...)
   step3 EPROM/FLASH autoselect
   step4 Program device (/Program[:switch])
   step5 Close of control program (/Close only together with parameter /Program)

Examples:

Example 1:

PG4UWcmd.exe /program:noanyquest /loadfile:c:\empfile.hex

Following operations will perform:

1. *start PG4UW.exe (if not already running)*
2. *load file c:\empfile.hex*
3. *start program device operation without questions*
4. *PG4UWcmd.exe is still running and periodically checking status of PG4UW.exe*
5. *when device programming completes, PG4UWcmd.exe is closed and is returning ExitCode depending on load file and device programming results in PG4UW.exe. When all operations were successful, PG4UWcmd.exe returns 0, otherwise returns value 1 or more.*

Example 2:

PG4UWcmd.exe /program:noanyquest /prj:c:\emproject.eprj

The operations are the same as in Example 1, just Load file operation is replaced by Load project file c:\emproject.eprj command.

Example 3:

Using PG4UWcmd.exe in batch file and testing return code of PG4UWcmd.exe.

```
rem ------- beginning of batch -------------------------
@echo off

rem Call application with wished parameters
PG4UWcmd.exe /program:noanyquest /prj:c:\emproject.eprj

rem Detect result of command line execution
rem Variable errorlevel is tested, value 1 or greater means the error occured

if errorlevel 1 goto FAILURE

echo Command line operation was successful
goto BATCHEND

:FAILURE
echo Command line operation error(s)

:BATCHEND
echo.
echo This is end of batch file (or continue)
pause
rem ------- end of batch -------------------------
```

Example 4:

Let's assume the PG4UW control program is running, and has user selected device. We need to load required data to PG4UW device buffer and save the selected device settings and buffer content to project file. Data required for device are stored in file c:\15001-25001\file_10.bin. Project file will be stored at *c:\projects\project_10.eprj*.

Following command line parameters should be specified to realize wished operation:

PG4UWcmd.exe /loadfile:c:\15001-25001\file_10.bin /saveproject:c:\projects\project_10.eprj

When PG4UW receives the commands, it will do following procedures:

1. *loads data file c:\15001-25001\file_10.bin*
2. *saves the currently selected device settings and buffer data to project file c:\projects\project_10.eprj*

If the result of operations performed is OK, PG4UWcmd application will return ExitCode (or ErrorLevel) value 0.
If there are some errors (can not load file or save to project file), PG4UWcmd application will return ExitCode value equal or greater than 1.

Note: When using the above commands, user must be sure the PG4UW is not performing any device operation, for example device programming. If the PG4UW is busy, it will refuse the commands and returns error status (ExitCode equal or greater than value 1).

7. Remote control of PG4UW applications running on different computer(s)

Remote control can be used to control PG4UW applications running on the same computer (localhost), and also PG4UW applications running on remote computer, that is connected to server computer by network. Server computer is computer, on which remote control program is running.

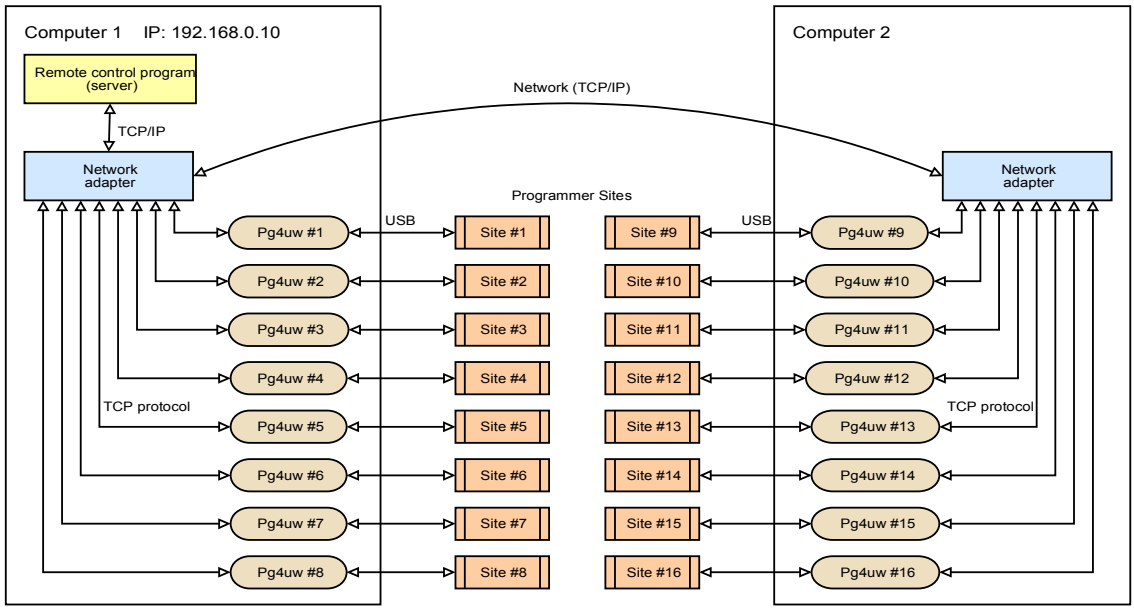
Following specials must be taken to consideration for PG4UW aplications running on remote computer:

1. Sites (PG4UW applications) can not be started directly from remote control program (server). They can be started manually on the computer where they are installed.
2. Address of server computer must be specified by command line parameter, when starting each PG4UW on remote computer. For more details look at **/remoteaddr:<address>** command line parameter description.
3. Remote operation command for "**Load project**" is sent to PG4UW applications by **SEND_CMD_LoadProject_CINDEX** remote command, so when using this for Sites running on different (remote) computer, the project file specified in the command must exist and be accessible on remote computer under the same project file path and name. Command "Load project" does not transfer any data of project file. It transfers just project file name.
4. Each Site must have its unique number, even if the Sites are connected and running on different computers. The valid numbers are from 1 to 16. In the future the range will be extended to higher values. Order number of Site and serial number of Site is specified by command line parameter **/usb:<index>:<programmer serial number>**.
5. Remote computer must be connected to server computer via network supporting TCP/IP protocol. Also firewalls must be configured to allow network access of both remote control application and PG4UW applications. Default port used for communication is 'telnet' (port value 23).

All other remote control commands are used in the same way, as for PG4UW clients running on the local computer with remote control program.

Situation when running Sites on more computers – localhost and remote computer - simultaneously is shown on following picture.

Picture 1. Typical configuration of remote controlled multiprogramming system running on two computers



The example of IP address for Computer 1 is 192.168.0.10. Computer 1 is also server computer, because Remote control program is running on it.

Description of command line parameter **/remoteaddr:<address>**

When we want to control programmer Sites (PG4UW clients) by remote control program (server), the PG4UW must know address of server computer. This address can be specified by command line parameter when starting PG4UW. The default value is 'localhost'. It means that PG4UW programs running on the same computer as remote control program, there is no need to specify the address. Address is required for PG4UW applications running on different computer only.

If we take configuration from Picture 1., we can tell that:

Sites #1 to #8, running on the server computer, do not need IP address to be specified.

Sites #9 to #16, running on remote computer, require IP address of server computer be specified in command line. For more details look at examples bellow.

Examples:

Example of using command line parameters for PG4UW applications started on server computer:

For Site #1:
PG4UW.exe /usb:1:469-00016 /enableremote:autoanswer
For Site #2:
PG4UW.exe /usb:2:469-00018 /enableremote:autoanswer
For Site #3:
PG4UW.exe /usb:3:469-00051 /enableremote:autoanswer
....
Numbers 469-00016, 469-00018 and 469-00051 are example serial numbers of programmer Sites.

Example of using command line parameters for PG4UW applications started on remote computer:

For Site #9:
PG4UW.exe /usb:9:469-00078 /enableremote:autoanswer /remoteaddr:192.168.0.10
For Site #10:
PG4UW.exe /usb:10:469-00079 /enableremote:autoanswer /remoteaddr:192.168.0.10
For Site #11:
PG4UW.exe /usb:11:469-00123 /enableremote:autoanswer /remoteaddr:192.168.0.10
....
Numbers 469-00078, 469-00079 and 469-00123 are example serial numbers of programmer Sites.
Server computer has IP address '192.168.0.10'.

These command line parameters are enough to use for successful connection to remote control program.

Tip: We recommend to use also Load project command line parameter /prj:<prj_name> when starting PG4UW programs. Project loading from command line parameter saves time, because PG4UW does not need to select default device after starting.

7.1 Sequence of steps is recommended when starting multiprogramming system

1. Be sure the programmer Sites on both computers are connected to USB ports and have correctly installed USB drivers.
   Information about hardware installation of programmers is available in programmer manuals.

2. **Start remote control program (server application)**
   When remote control program (RCP) is started, it waits for Clients (PG4UW). When client with proper index is started, it will automatically connect to RCP. After establishing connection between client-server, remote control commands can used.

3. **Start Sites #1 to #8, with proper indexes and serial numbers of Sites specified.**
   Starting of Sites #1 to #8 can be performed automatically by remote control program, or manually by running PG4UW #1 to PG4UW #8 with proper parameters in command line. There can be started the same PG4UW.exe application for each Site. The only difference will be in command line parameters.

4. **Start Sites #9 to #16 on remote computer.**
   Sites can be started manually by running PG4UW #9 to #16 with suitable command line parameters. Of course PG4UW must be installed on the remote computer. When sites are connected, the remote control program should detect new clients connected and can make suitable indication of this information.

This sequence can be practically tested with remote control demonstration program 'Multidemo' included in standard installation of PG4UW. Program Multidemo is windows based application written in Borland Delphi programming environment. There is available also version of Multidemo written in Microsoft Visual Studio .NET. Complete sources of this program are included too.

7.2 Basic description of Multidemo application

Program Multidemo is included with commented sources written for Borland Delphi 6 or newer. The main conception of the program is to make control of one or more PG4UW applications

- communication between Multidemo and PG4UW applications is made by TCP/IP protocol on port 'telnet' (value 23)
- Multidemo acts as server and PG4UW applications act as clients
- Multidemo can control PG4UW running on the same computer (localhost) and also on different (remote) computer connected together by network with TCP/IP support
- remote control commands are send/received in form of strings
- to simplify the usage of remote control, we made library remotelb.dll, which contains functions and procedures to send commands from server to PG4UW clients
- status information received from PG4UW is proceed in server Multidemo remote control program by following scheme:

1. PG4UW status information strings are stored in internal FIFO in library remotelb.dll
2. Communication between client-server is made mostly asynchronously
3. To make request of information from PG4UW, remote control program has to send command procedure **SEND_CMD_GetProgStatus_CINDEX**(index) to all connected PG4UW applications (clients).
4. To pick the information received from PG4UW clients, function **GetCommandStringFromFIFO_CINDEX**(index) must be called for each connected client (PG4UW) in remote control program
5. Program Multidemo uses procedure **ProcessProcByFIFO**(index) to receive and process received information from PG4UW client with index 'index'.
   Procedure **ProcessProcByFIFO**(index) uses following functions:
   - function **GetCommandStringFromFIFO_CINDEX**(index) to receive info from PG4UW
   - procedure **intServerProcessProc**(received_cmd:) is used to decode received information string and process decoded information

6. To ensure all received information to be proceed, **ProcessProcByFIFO**(index) is called periodically in timer OnTimer event.
   Recommended interval of timer tick event is 100-1000ms. Calling of both important routines is made in timer event.
   In the first timer tick, **SEND_CMD_GetProgStatus_CINDEX**(index) is called for all connected PG4UW.
   In the second timer tick, received information is proceed by **ProcessProcByFIFO**(index) for all connected PG4UW.
   And this sequence is repeated periodically.

8. .NET support

Remote control support for .NET applications is available by library remotelbNET.dll. The library contains many of remote control methods available by native remotelb.dll library file.

Source files of remotelbNET and also of other .NET examples (demos) are available in folder, where all other remote control examples are included in.

The typical path to remote control examples looks like this:

C:\Program Files\Elnec_sw\Programmer\remotectrl\programs

9. System requirements

9.1 System requirements for remote control program

Minimum PC system requirements for remote control application:

- Microsoft Windows® 98, 2000, XP or later
- PC Pentium 233
- 32 MB of RAM
- 10 MB of free disk space

Recommended PC system requirements for remote control application:

- Microsoft Windows® XP Professional
- PC Pentium III 600
- 128 MB of RAM
- 10 MB of free disk space

9.2 System requirements for application PG4UW when using remote control function

Minimum PC system requirements:

- Microsoft Windows® 98, 2000, XP or later
- PC Pentium 4
- 256 MB of RAM
- 200 MB of free disk space
- LPT printer port (for programmers connected via LPT port)
- USB port ver. 1.1 or later (for programmers connected via USB port)

Recommended PC system requirements:

- Microsoft Windows® XP Professional
- PC Pentium Core 2 Duo
- 512 MB of RAM
- 1000 MB of free disk space
- LPT printer port supporting EPP/ECP modes (for programmers connected via LPT port)
- USB port ver. 1.1 or later (for programmers connected via USB port)

9.3 Common requirements

Both remote control program and PG4UW program need network adapter with TCP protocol support installed. The network adapter can be virtual (Microsoft loopback adapter) or real network adapter (network card with proper drivers installed).

When using remote control program and PG4UW program on the same computer, there is no need to have real network adapter (network card) installed.

When using remote control program and PG4UW program on the remote computers, real network adapter (network card) has to be installed on each computer.

Installation of virtual network adapter (example for Windows XP):

1. click Start menu
2. select **Settings** item
3. select **Control panel**
4. in Control panel window click on **Add Hardware**
5. in **Add Hardware Wizard** dialog click on button Next
6. on question “Have you already connected this hardware on computer?” select option “Yes, I have already connected the hardware” and click on button Next
7. in the list of available hardware select the item “Add a new hardware device” and click on button Next
8. on question “What do you want the wizard to do?” select option “Install the hardware that I manually select from list (Advanced)” and click button Next
9. from the list “Common hardware types” select item “Network adapters” and click button Next
10. from left panel “Manufacturer” select “Microsoft”
11. from right panel “Network adapter” select “Microsoft Loopback adapter” and click button Next
12. on the confirmation click button Next again
13. the adapter is installed

Now you can configure the adapter to support TCP protocol. Configuration is made in the same way as for real network cards.

10. Version history of this documentation

Ver. 1.39 (Feb 16th 2017)
- removed text about registration of ISP programming and license file keyfile.val

Ver. 1.38 (May 31st 2016)
- added more information to command ine parameter /enableremote:autoanswer (chapter “6.1 Configuration command line parameters”)

Ver. 1.36 (March 9th 2016)
- just replaced Pg4uw by PG4UW (matching case)

Ver. 1.35 (March 6th 2015)
- added information about new functions added to remotelb.dll and remotelbNET.dll:
  SEND_CMD_CreditBox_GetTotalAvailableCredits and
  SEND_CMD_CreditBox_GetTotalAvailableCredits_CINDEX

Ver. 1.34 (August 22nd 2014)
- added information about new functions added to remotelb.dll and remotelbNET.dll:
  SEND_CMD_GET_PG4UW_VERSION and SEND_CMD_GET_PG4UW_VERSION_CINDEX

Ver. 1.33 (June 13th 2014)
– removed paragraph about availability of Turbo Delphi free download.

Ver. 1.32 (May 13th 2014)
- added information about new “client send operation result” command:
  TCP_CMD_DETAILED_OPRESULT

Ver. 1.31 (November 13th 2013)
– updated restrictions in Chapter 5
– corrected grammar “thread save → thread safe”

Ver. 1.30 (September 19th 2013)
– formatting changes in document, Elnec logo applied to header

Ver. 1.29 (March 5th 2013)
– added information about message TCP_CMD_SERVER_CAN_RUN_ANOTHER_SITE in chapter 5 'Using remote control with multiply programmers (multiprogramming)'

Ver. 1.28 (June 28th 2012)
– added note about programmer serial number into chapter 5 'Using remote control with multiply programmers (multiprogramming)'

Ver. 1.27 (June 14th 2011)
– added restrictions about “Using remote control with multiply programmers (multiprogramming)”

Ver. 1.26 (April 5th 2011)

– added source code examples

Ver. 1.25 (March 03rd 2011)

– added information about new remote control function SEND_CMD_GetProjectFileChecksum and minimal requirements modification

Ver. 1.24 (August 5th 2010)

– section “Using remote control with multiply programmers (multiprogramming)”, Restrictions

section, added BeeHive204 and BeeProg2 programmers

Ver. 1.23 (January 9th 2009)
– added restrictions about “Using remote control with multiply programmers (multiprogramming)”

Ver. 1.22 (November 15th 2007)

– added information about new multiprogramming remote control example sources “MultiDemo.VB.NET” for Microsoft Visual Studio .NET

Ver. 1.21 (September 17th 2007)
– added information about “Remote control of PG4UW applications running on different computer(s)”
– added information about new command line parameter */startvisible*

Ver. 1.20 (July 26th 2007)

– added information about command line parameter /enableremote:autoanswer

Ver. 1.19 (March 15th 2007)

- added information about new remote control examples written in Microsoft Visual Basic .NET

Ver. 1.18 (February 22nd 2007)

- added Note 3. about null-terminated strings in description of functions
  GetCommandStringFromFIFO and GetCommandStringFromFIFO_CINDEX

Ver. 1.17 (February 15th 2007)
- added information about new command line parameter /Demo
- added information about new remote control functions
  GetCommandStringFromFIFO and GetCommandStringFromFIFO_CINDEX
- added new information about callback procedure parameters, especially parameter 'vProcessProc' used in procedures:
  CreateServerAndMakeListenToClients and
  CreateServerAndMakeListenToMultiClients

Ver. 1.16 (December 6th 2006)

- added information about new command line parameter /close:always
- changed execution order of command line parameters /prj:... and /loadfile:...
  /Prj command is executed before /loadfile command.

Ver. 1.15 (November 16th 2006)

- added information about new command line parameter /saveproject:...

Ver. 1.14 (October 18th 2006)

- added information about new remote control examples written in Microsoft Visual Basic 6

Ver. 1.13 (September 29th 2006)

- added additional information about /Eprom_Flash_Autoselect command line parameter

Ver. 1.12 (August 24th 2006)

- added information about new multiprogramming remote control example sources “MultiDemo”
- added description of new remote command function
  SEND_CMD_GetProgrammerConnectedStatus

Ver. 1.11 (July 12th 2006)

- added description of new remote command function SEND_CMD_GetDev_Checksum

Ver. 1.10 (June 5th 2006)

- added new chapter about “Using remote control with multiply programmers (multiprogramming)”

Ver. 1.09 (April 10th 2006)

- modified “Recommended PC system requirements” to more powerful configuration

Ver. 1.08 (March 28th 2006)

- added description of remote control examples added in the form of source files
- added description of remote command functions:
  SEND_CMD_ShowMainForm, SEND_CMD_HideMainForm and SEND_CMD_CloseApp

Ver. 1.07 (March 6th 2006)

- start of using the version history list
- added description of new remote command function SEND_CMD_SaveFile

END of documentation