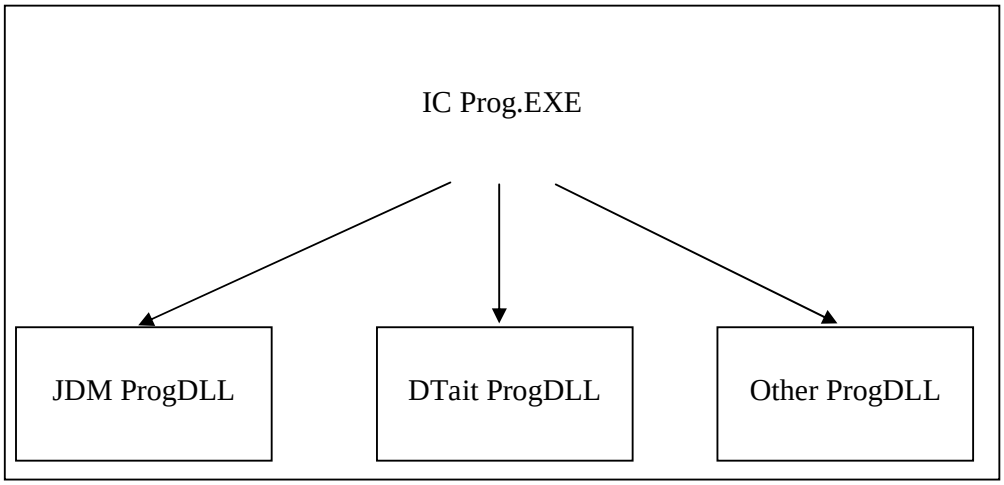
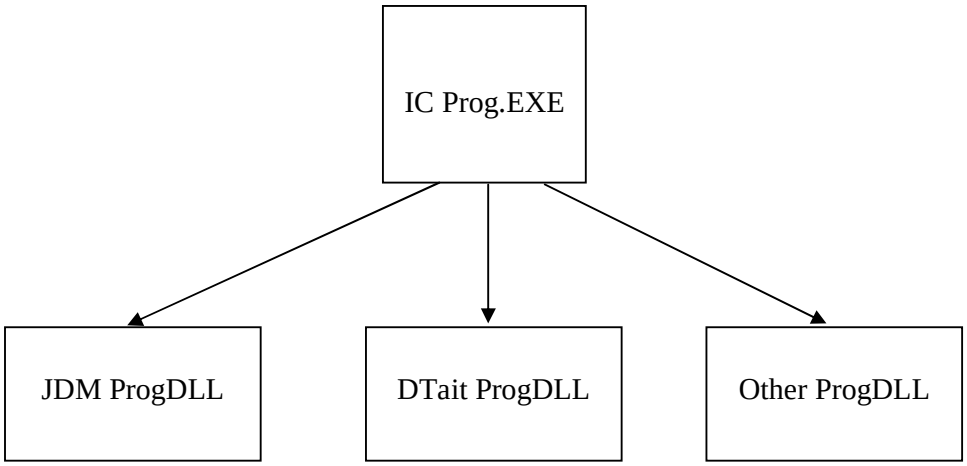
ICProg Programmers Plugin Engine



The ICProg application is actually a single monolythique executable that must be reworked every time in order to support new programmers hardwares.

The present document aims to provide a solution to make interfacing ICProg to new hardwares more flexible and easier.

The ICProg Plugin Engine:
The solution is to separate ICProg code from the hardware management and interfacing code.
This can be done by using the Dynamic Linking Libraries (DLL). The used/proposed technique is easy to implement in ICProg. It's based on what's called LATE BINDING to DLLs (or Dynamic load of DLLs).



To get it work we must first of all define the responsibilities of every part of the software:

1- ICProg.EXE :
   a. Manages all Chips Informations
   b. Manages Loading and Writing Hex Files
   c. Execute Writing/Reading/Erasing algorithms for every chipset using theses Signals (some of these signals can be unused or optional):
      i. DataIN
      ii. DataOut
      iii. Clock
      iv. MCLR
      v. Vpp
      vi. Vcc
   d. Contains code to Identify and ENUMERATE all DLLs that control the hardware that generate previous signals
   e. Contains code to Select and LOAD a specified DLL to take control of a programming hardware (programmer)

2- Prog DLL:
   a. Manages All interactions with a programmer hardware
   b. Contains code to Communicate with Hardware in order to Set/Clear/Read these signals (same signals as ICProg)
      i. DataIN
      ii. DataOut
      iii. Clock
      iv. MCLR
      v. Vpp
      vi. Vcc

   c. Provide the user with an interface (forms and other stuff) to configure Hardware
   d. Save/Load hardware settings

From this tasks separation, we can see that ICProg will never interacts with hardware directly. It will LOAD a DLL that was written to control a specific hardware (ProgDLL). That means that ICProg will not use any

driver of any kind, nor will it contain vcl components to control the pc Ports. All this stuff is the ProgDLL's responsibility now.
ICProg now know how to execute Algorithms with signals HI/LO, nothing else. He Loads a DLL, connects to it, and call DataIn to read a signal state, if the DLL is a JDMProgDll it will read for example the TXIn signal state, if it's a DTaitProgDLL it will read the Busy signal of DB25 Parallel port, if it's a NewI2CProg invented it will say to the I2CDLL to send a message to the programmer to read In data … and so on …

Well, from that we see that ICProg must ALWAYS see the same interface from the progDlls, he must be able to call always the same functions/procedures with the same signatures (number and type of parameters).
For that we have defined a STANDARD DLL INTERFACE that every ProgDLL MUST provide to be usable with (loadeable in) icProg.

ProgDLLs INTERFACE :

Every dll MUST implement these set of functions/procedures:

```
IsICProgAddon,
GetProgrammerName,
InitDll,
DoneDll,
SetupProgrammer,
SetMCLR,
SetDataOut,
SetClock,
GetData,
PowerOn,
PowerOff;
```

```
function IsICProgAddon() : boolean; stdcall;
function GetProgrammerName() : Pchar; stdcall;
function InitDll() : boolean; stdcall;
function DoneDll() : boolean; stdcall;
function SetupProgrammer(AppHandle : THandle) : integer; stdcall;
procedure SetMCLR( aState : Integer); stdcall; // 0 = Low, 1 = High
procedure SetDataOut( aState : Integer); stdcall; // 0 = Low, 1 = High
procedure SetClock( aState : Integer); stdcall; // 0 = Low, 1 = High
function GetData() : integer; stdcall;
procedure PowerOn(); stdcall;
procedure PowerOff(); stdcall;
```

Description:

```
function IsICProgAddon() : boolean; stdcall;
begin
  result := True;
end;
```

This function MUST exists and return True. It's used by ICProg to determine if the DLL is an ICProg plugin or not.

```
function GetProgrammerName() : Pchar; stdcall;
begin
  result := Pchar(PROG_NAME);
end;
```

This Function returns the LITERAL NAME of the programmer, the name that will be shown by icProg in its menu (PopupMenuItem.caption).
PS. Note that this function return a PChar instead of String because the Strings have lot of issues when they are to be used in DLLs as parameters or return values of Exported functions and procedures.

```
function InitDll() : boolean; stdcall;
begin
  // do init stuff (global vars can be initialised/allocated here)
  result := True;
end;
```

This function is first called by ICProg just after loading the DLL, this function is responsible of initializing the dll, creating global classes of the DLL, initializing global variables… For example, for the JDM programmer, if we are using the SmallPort component to access com port, this routine can create the global instance of TSmallPort class, it can scan and create the list of com ports (to be shown in the setting hardware form) and so on.
The return value of this function determine if there where problems or not in initializing the DLL.

```
function DoneDll() : boolean; stdcall;
begin
  // do prepare to quit DLL (Global vars can be Freed here)
  result := True;
end;
```

This function do the reverse of InitDLL, it means that it release allocated instances of classes, closes communication ports if needed …

```
function SetupProgrammer(AppHandle : THandle) : integer; stdcall;
var FrmProgSetup : TfrmJDMProgSetting;
    oldAppHnd    : THandle;
begin
  //PS.  we must pass the calling application Handle because the DLL and the Main
  //Program does not share the same TApplication object, so to make the ProgSetting
  //Form MODAL we must pass it the Main Application Handle as Owner
  oldAppHnd := Application.Handle;
  Application.Handle := AppHandle;
  FrmProgSetup := TfrmJDMProgSetting.Create(Application);
  try
    if FrmProgSetup.ShowModal = mrOK then begin
      ShowMessage('Config accepted and Saved');
      result := 0;
    end
    else begin
      ShowMessage('You cancelled the Config Box');
      result := -1;
    end;
  finally
    FrmProgSetup.Free;
    Application.Handle := oldAppHnd;
  end;
end;
```

This function will be called by ICProg to give the user the possibility to configure the hardware.
P.S Note that this function accepts as parameter the handle of the calling application. This is necessary because (of Delphi) the TApplication class' instance of the ICProg Application is different from the instance created in the DLL, and in order to have a correct behaviour of Modal forms, we must pass the Caller TApplication instance as the owner of the created form.

```
procedure SetMCLR( aState : Integer); stdcall; // 0 = Low, 1 = High
begin
end;
```
This procedure Sets/Clears the MCRL signal.

```
procedure SetDataOut( aState : Integer); stdcall; // 0 = Low, 1 = High
begin
end;
```
This procedure Sets/Clears the DataOut signal.

```
procedure SetClock( aState : Integer); stdcall; // 0 = Low, 1 = High
begin
end;
```
This procedure Sets/Clears the Clock signal.

```
function GetData() : integer; stdcall;
begin
  result := 1; //get data pin
end;
```
This function read the state of the data Signal.

```
procedure PowerOn(); stdcall;
begin
end;
```
This procedure Sets the power On.

```
procedure PowerOff(); stdcall;
begin
end;
```
This procedure Sets the power Off.

PS. I have separated the PowerOn/PowerOff procedures instead of making just a simple SetPower(state) procedure, because some hardwares can use two different signals to set/clear power (by using two state or three state gates…)

How ICProg uses this DLL ?

After loading the DLL (we will see this later in this document) ICProg must :

1- Call InitDll(…) // the DLL creates its internal vars and loads its saved settings
…
2- it can Call ProgrammerSetup(…) to give the user the possibility to setup his Hardware

3- Call Other functions …. (VPP… MCLR….)
4- Call DoneDll(…) // To destroy allocated vars ….

PS. Note that it's the responsibility of the DLL to Save/Load the hardware setting (my be using a special registry key… remember that the keys must be unique for each programmerDLL).

The ICProg Side:

In order to give ICProg access to the ProgDlls, I have written a simple Delphi UNIT named "**unProgDlls.pas**" .

This unit declares some GLOBAL functions and procedures, and a global class.

The global class

ICProgrammers : TICProgrammers;

This class instance is created in the initialization section of the unit and destroyed in the finalization section.
This class just holds the list of the founds programmers DLLs. ICProgs fills it using the procedure FindProgDlls(const aPath : string).
To know the programmers DLL found by icProg in the aPath (passed as param to FindProgDlls) just read the property ICProgrammers.ProgList.Count
To access any programmer just access to

ICProgrammers[index].ProgName ➔ returns the Name of the programmer
ICProgrammers[index].ProgDllName ➔ returns the Name of DLL of the programmer

We will see an example later.

Global Functions and procedures:

```
InitDll       : TInitFunc   = nil;
DoneDll       : TInitFunc   = nil;
SetupProgrammer: TSetupProgFunc = nil;
SetMCLR       : TPinSetProc = nil;
SetDataOut    : TPinSetProc = nil;
SetClock      : TPinSetProc = nil;
GetData       : TPinGetFunc = nil;
PowerOn       : TPowerProc  = nil;
PowerOff      : TPowerProc  = nil;
IsICProgAddon : TBoolFunc   = nil;
GetProgrammerName : TProgNameFunc = nil;
```

These are Pointers on procedures/functions, they are used once they are connected to DLL exported functions/procedures.

function LoadProgDll(const aDllName : string) : boolean;
This function loads (connects) the dll to icProg (aDllName must be the name of the dll file for example "jdmdll.dll" or "c:\toto\jdmprog.dll"). if it succeeds the result will be True.

function IsProgDll(const aDllName : string; var aProgName : string) : boolean;

This function tests aDllName dll, and if it's an icProg plugin it returns the name of the programmer in aProgName.

procedure FindProgDlls(const aPath : string);
This procedure scans aPath for all the DLLs, and if it finds a dll it register (adds) it in the global class ICProgrammers.

Example of Use :
Lets suppose that we have two ProgDlls named :
JDMProg.DLL for the "JDM Programmer"
DTaitSerialProg.DLL for the "David Tait Serial Programmer"

In ICProg these programmers are used (one on COM port, the other on the LPT port).

In ICProg we have just to do the following:

Step 1: Add the unit unProgDlls.pas to the main form Uses section.
When ICProg starts, the global class instance ICProgrammers is created in the initialization section.

When icProg starts, lets suppose that the progsDLLs are copied in the ICProg.EXE directory, so we can access to their directory using ExtractFilePath(Application.ExeName)

ICProg has to do the folowing :
Step 2: Enumerate all the ProgsDlls
FindProgDlls(ExtractFilePath(Application.ExeName));
This will automatically scan the Exe file directory and adds these two entries
ICProgrammers[0] => (ProgName:"JDM Programmer", ProgDLL:"JDMProgDLL")
ICProgrammers[1] => (ProgName:" David Tait Serial Programmer", ProgDLL:" DTaitSerialProg.DLL")

Step3: We will give the user a way to select a programmer and we retrieve the index of the selected programmer
Step4: ICProg Loads the concerned ProgDLLs
LoadProgDll(ICProgrammers[indx].ProgDll);

Step 5: ICProg is ready to use the functions of the prog DLL

```
function IsICProgAddon() : boolean; stdcall;
function GetProgrammerName() : Pchar; stdcall;
function InitDll() : boolean; stdcall;
function DoneDll() : boolean; stdcall;
function SetupProgrammer(AppHandle : THandle) : integer; stdcall;
procedure SetMCLR( aState : Integer); stdcall; // 0 = Low, 1 = High
procedure SetDataOut( aState : Integer); stdcall; // 0 = Low, 1 = High
procedure SetClock( aState : Integer); stdcall; // 0 = Low, 1 = High
function GetData() : integer; stdcall;
procedure PowerOn(); stdcall;
procedure PowerOff(); stdcall;
```

Step6: the user can choose to configure his programmer (mybe a menu option), ICProg calls
SetupProgrammer(application.Handle);

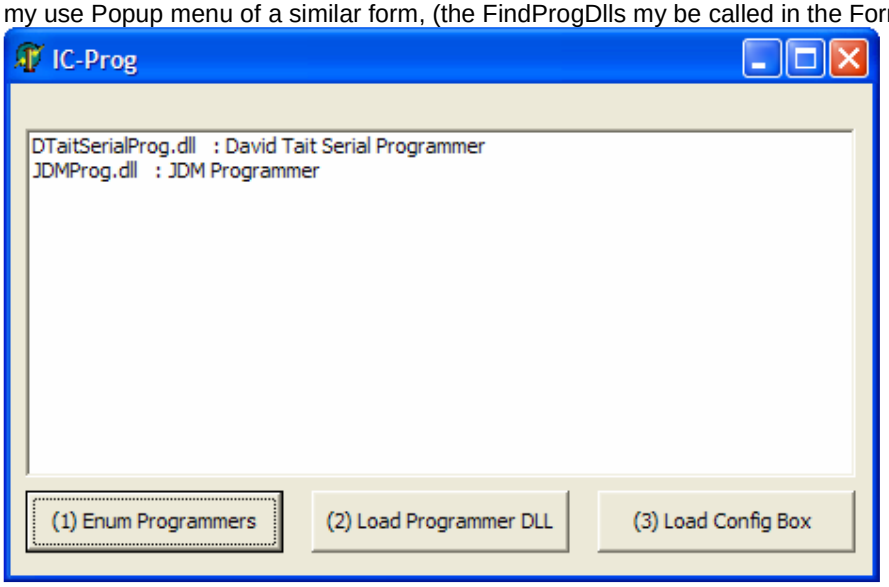
Given sources codes:
All my codes where created and tested using Delphi 6 (if you want it for D5 or D7 let me know I'll try to adapt it).

I have created a project group named "ICProg Group.bpg"
It contains 3 sub-projects:
Project1- Icprog.exe : this is the ICProg Project that uses the previous units
It uses (contains) unProgDLLs.pas and an example of use ufMain.DFM/frmMain.pas

Project2- JDMProgDLL.DPR
The project (dpr) contains the sources codes of the JDMProgrammer (there is no hardware management, it's just a template for a real progdll).
This project uses the unJDMProgSettings DFM to propose a form for setting the JDM programmer.

Project3- DTaitSerialProgDLL.DPR
The project (dpr) contains the sources codes of the David Tait Serial Programmer (there is no hardware management, it's just a template for a real progdll).
This project uses the unDTaitSerialProgSettings DFM to propose a form for setting the DTait programmer.

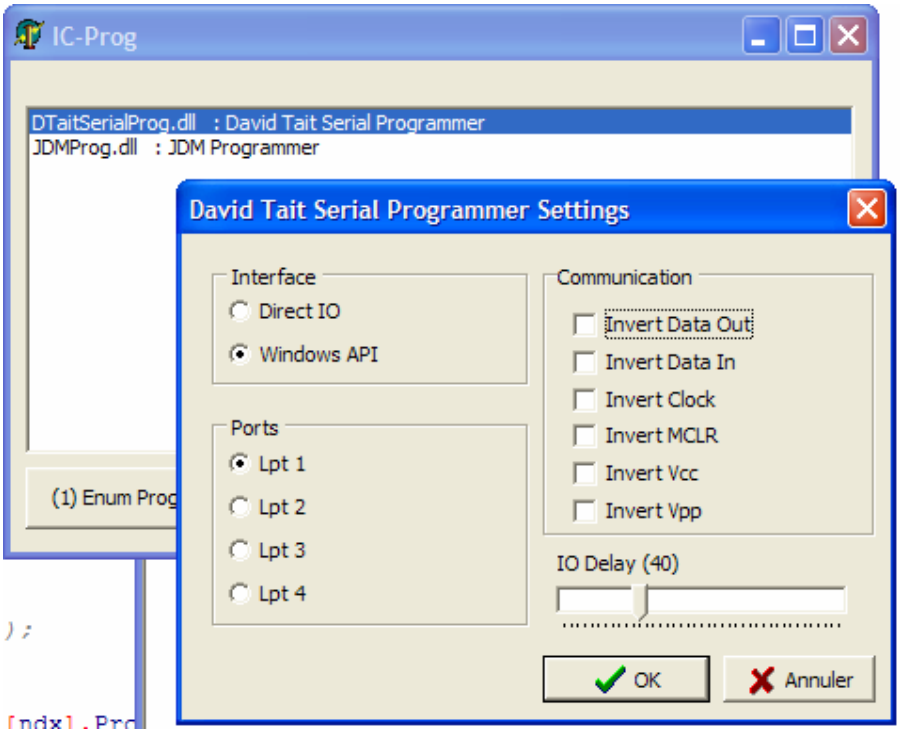
USE CASE:
The main program propose a simple case of use, using a List Box as tool to choose a programmer, icProg my use Popup menu of a similar form, (the FindProgDlls my be called in the FormCreate event).



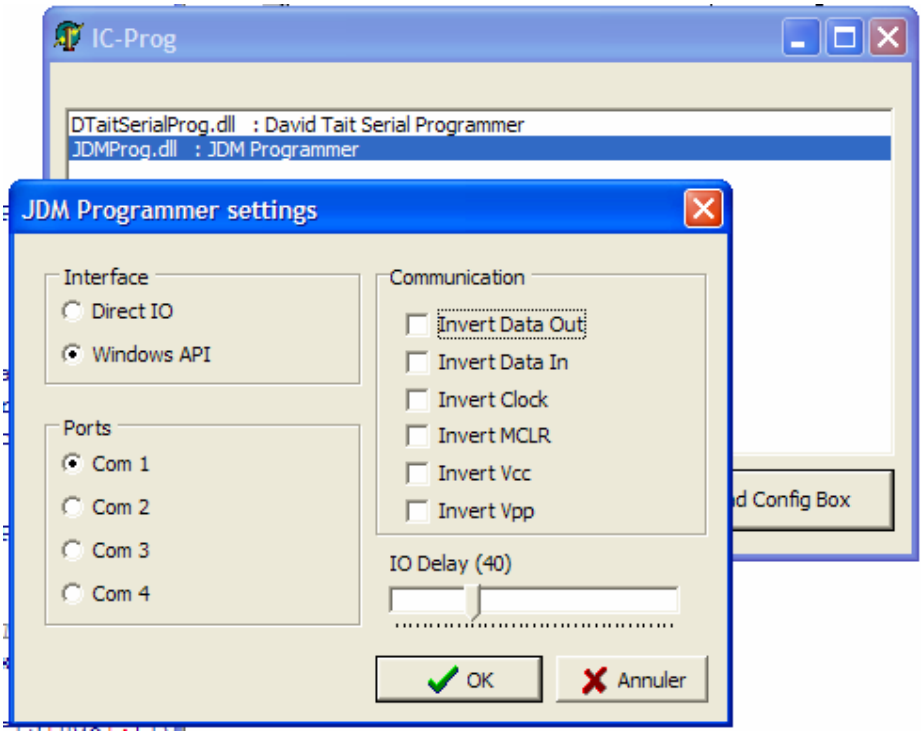
The user selects a programmer, (normally in icProg this should call automatically LoadProgDll function)

The user can click on progSetting button (or menu option) to see the specific programmer ConfigForm.

DTait Config Screen



JDM Config Screen



ICProg can use the function MCLR/VPP ….

Note : in this example each time you selects a programmer you MUST click on the LOAD prog button (middle button) to unload the previous prog and load the new dll.

Note :

- New Programmers have only to develop a ProgDLL.DLL and copy it to icProg dlls path.
- It's the progDll responsibility to provide drivers and vxd to access ports under WNT/2K/XP

If any need to help let me know.

Ahmed LAZREG
ahmedlazreg@free.fr
alazreg@yahoo.fr