

```

1: { SdpoSerial v0.1.4
2:
3:   Copyright (C) 2006-2010 Paulo Costa
4:
5:   This library is Free software; you can redistribute it and/or modify it
6:   under the terms of the GNU Library General Public License as published by
7:   the Free Software Foundation; either version 2 of the License, or (at your
8:   option) any later version.
9:
10:  This program is distributed in the hope that it will be useful, but WITHOUT
11:  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
12:  FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License
13:  for more details.
14:
15:  You should have received a Copy of the GNU Library General Public License
16:  along with This library; if not, Write to the Free Software Foundation,
17:  Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
18:
19:  This license has been modified. See File LICENSE.ADDON for more information.
20:  Should you find these sources without a LICENSE File, please contact
21:  me at paco@fe.up.pt
22: }
23:
24: unit SdpoSerial;
25:
26: {$mode objfpc}{$H+}
27:
28: interface
29:
30: uses
31:   {$IFDEF LINUX}
32:     Classes,
33:   {$IFDEF UseCThreads}
34:     cthreads,
35:   {$ENDIF}
36:   {$ELSE}
37:     Windows, Classes, //registry,
38:   {$ENDIF}
39:   SysUtils, synaser, LResources, Forms, Controls, Graphics, Dialogs;
40:
41:
42: type
43:   TBaudRate=(br__300, br__600, br__1200, br__2400, br__4800, br__9600, br_19200,
44:               br_38400, br_57600, br115200, br230400, br460800, br921600);
45:   TDataBits=(db8bits,db7bits,db6bits,db5bits);
46:   TParity=(pNone,pOdd,pEven,pMark,pSpace);
47:   TFlowControl=(fcNone,fcXonXoff,fcHardware);
48:   TStopBits=(sbOne,sbTwo);
49:
50:   TModemSignal = (msRI,msCD,msCTS,msDSR);
51:   TModemSignals = Set of TModemSignal;
52:
53: const
54:   ConstsBaud: array[TBaudRate] of integer=(
55:     300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200,
56:     230400, 460800, 921600);
57:
58:   ConstsBits: array[TDataBits] of integer=(8, 7, 6, 5);
59:   ConstsParity: array[TParity] of char=('N', 'O', 'E', 'M', 'S');
60:   ConstsStopBits: array[TStopBits] of integer=(SB1, SB2);
61:
62:

```

```

63: type
64:     TSdpoSerial = class;
65:
66:     TComPortReadThread=class(TThread)
67:     public
68:         MustDie: boolean;
69:         Owner: TSdpoSerial;
70:     protected
71:         procedure CallEvent;
72:         procedure Execute; override;
73:     published
74:         property Terminated;
75:     end;
76:
77:     { TSdpoSerial }
78:
79: TSdpoSerial = class(TComponent)
80: private
81:     FActive: boolean;
82:     FSynSer: TBlockSerial;
83:     FDevice: string;
84:
85:     FBaudRate: TBaudRate;
86:     FDataBits: TDataBits;
87:     FParity: TParity;
88:     FStopBits: TStopBits;
89:
90:     FSoftflow, FHardflow: boolean;
91:     FFlowControl: TFlowControl;
92:
93:     FOnRxData: TNotifyEvent;
94:     ReadThread: TComPortReadThread;
95:
96:     FAltBaudRate: integer;
97:
98:     procedure DeviceOpen;
99:     procedure DeviceClose;
100:
101:     procedure ComException(str: string);
102:     function BaudRateValue: integer;
103:
104: protected
105:     procedure SetActive(state: boolean);
106:     procedure SetBaudRate(br: TBaudRate);
107:     procedure SetAltBaudRate(altbr: integer);
108:     procedure SetDataBits(db: TDataBits);
109:     procedure SetParity(pr: TParity);
110:     procedure SetFlowControl(fc: TFlowControl);
111:     procedure SetStopBits(sb: TStopBits);
112:
113: public
114:     constructor Create(AOwner: TComponent); override;
115:     destructor Destroy; override;
116:
117:     procedure Open;
118:     procedure Close;
119:
120:     // read data from port
121:     function DataAvailable: boolean;
122:     function ReadData: string;
123: //     function ReadBuffer(var buf; size: integer): integer;
124:

```

```

125:         // write data to port
126:     function WriteData(data: string): integer;
127:     function WriteBuffer(var buf; size: integer): integer;
128:
129:     // read pin states
130:     function ModemSignals: TModemSignals;
131:     function GetDSR: boolean;
132:     function GetCTS: boolean;
133:     function GetRing: boolean;
134:     function GetCarrier: boolean;
135:
136:     // set pin states
137: //     procedure SetRTSDTR(RtsState, DtrState: boolean);
138:     procedure SetDTR(OnOff: boolean);
139:     procedure SetRTS(OnOff: boolean);
140:     procedure SetBreak(OnOff: boolean);
141:
142:     published
143:         property Active: boolean read FActive write SetActive;
144:
145:         property BaudRate: TBaudRate read FBaudRate write SetBaudRate; // default br115200;
146:         property AltBaudRate: integer read FAltBaudRate write SetAltBaudRate; // default br115200
147:     ;
148:     property DataBits: TDataBits read FDataBits write SetDataBits;
149:     property Parity: TParity read FParity write SetParity;
150:     property FlowControl: TFlowControl read FFlowControl write SetFlowControl;
151:     property StopBits: TStopBits read FStopBits write SetStopBits;
152:
153:     property SynSer: TBlockSerial read FSynSer write FSynSer;
154:     property Device: string read FDevice write FDevice;
155:
156:     property OnRxData: TNotifyEvent read FOnRxData write FOnRxData;
157: end;
158: procedure Register;
159:
160: implementation
161:
162: { TSdpoSerial }
163:
164: procedure TSdpoSerial.Close;
165: begin
166:     Active:=false;
167: end;
168:
169: procedure TSdpoSerial.DeviceClose;
170: begin
171:     // flush device
172:     if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
173:         FSynSer.Flush;
174:         FSynSer.Purge;
175:     end;
176:
177:     // stop capture thread
178:     if ReadThread<>nil then begin
179:         ReadThread.FreeOnTerminate:=false;
180:         ReadThread.MustDie:= true;
181:         while not ReadThread.Terminated do begin
182:             Application.ProcessMessages;
183:         end;
184:         ReadThread.Free;
185:         ReadThread:=nil;

```

```

186:     end;
187:
188:     // close device
189:     if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
190:         FSynSer.Flush;
191:         FSynSer.CloseSocket;
192:     end;
193: end;
194:
195: constructor TSdpoSerial.Create(AOwner: TComponent);
196: begin
197:     inherited;
198:     //FHandle:=-1;
199:     ReadThread:=nil;
200:     FSynSer:=TBlockSerial.Create;
201:     FSynSer.LinuxLock:=false;
202:     FHardflow:=false;
203:     FSoftflow:=false;
204:     FFlowControl:=fcNone;
205:     {$IFDEF LINUX}
206:     FDevice:='/dev/ttyS0';
207:     {$ELSE}
208:     FDevice:='COM1';
209:     {$ENDIF}
210:     // FBaudRate:=br115200;
211:     FAltBaudRate := 0;
212: end;
213:
214: function TSdpoSerial.DataAvailable: boolean;
215: begin
216:     if FSynSer.Handle=INVALID_HANDLE_VALUE then begin
217:         result:=false;
218:         exit;
219:     end;
220:     result:=FSynSer.CanReadEx(0);
221: end;
222:
223: destructor TSdpoSerial.Destroy;
224: begin
225:     Close;
226:     FSynSer.Free;
227:     inherited;
228: end;
229:
230: procedure TSdpoSerial.Open;
231: begin
232:     Active:=true;
233: end;
234:
235: procedure TSdpoSerial.DeviceOpen;
236: begin
237:     FSynSer.Connect(FDevice);
238:     if FSynSer.Handle=INVALID_HANDLE_VALUE then
239:         raise Exception.Create('Could not open device '+ FSynSer.Device);
240:
241:     FSynSer.Config(BaudRateValue(),
242:                   ConstsBits[FDataBits],
243:                   ConstsParity[FParity],
244:                   ConstsStopBits[FStopBits],
245:                   FSoftflow, FHardflow);
246:
247:     // Launch Thread

```

```
248:   ReadThread := TComPortReadThread.Create(true);
249:   ReadThread.Owner := Self;
250:   ReadThread.MustDie := false;
251:   ReadThread.start;
252: end;
253:
254:
255: function TSdpoSerial.ReadData: string;
256: begin
257:   result:='';
258:   if FSynSer.Handle=INVALID_HANDLE_VALUE then
259:     ComException('can not read from a closed port.');
```

260:

```
261:   result:=FSynSer.RecvPacket(0);
262: end;
263:
264: procedure TSdpoSerial.SetActive(state: boolean);
265: begin
266:   if state=FActive then exit;
267:
268:   if state then DeviceOpen
269:   else DeviceClose;
270:
271:   FActive:=state;
272: end;
273:
274: procedure TSdpoSerial.SetBaudRate(br: TBaudRate);
275: begin
276:   if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
277:     FSynSer.Config(BaudRateValue(), ConstsBits[FDataBits], ConstsParity[FParity],
278:       ConstsStopBits[FStopBits], FSoftflow, FHardflow);
279:   end;
280:   FBaudRate:=br;
281: end;
282:
283: procedure TSdpoSerial.SetAltBaudRate(altbr: integer);
284: begin
285:   FAltBaudRate := altbr;
286:   if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
287:     FSynSer.Config(BaudRateValue(), ConstsBits[FDataBits], ConstsParity[FParity],
288:       ConstsStopBits[FStopBits], FSoftflow, FHardflow);
289:   end;
290: end;
291:
292:
293: procedure TSdpoSerial.SetDataBits(db: TDataBits);
294: begin
295:   if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
296:     FSynSer.Config(BaudRateValue(), ConstsBits[FDataBits], ConstsParity[FParity],
297:       ConstsStopBits[FStopBits], FSoftflow, FHardflow);
298:   end;
299:   FDataBits:=db;
300: end;
301:
302: procedure TSdpoSerial.SetFlowControl(fc: TFlowControl);
303: begin
304:   if fc=fcNone then begin
305:     FSoftflow:=false;
306:     FHardflow:=false;
307:   end else if fc=fcXonXoff then begin
308:     FSoftflow:=true;
309:     FHardflow:=false;
```

```

310: end else if fc=fcHardware then begin
311:     FSoftflow:=false;
312:     FHardflow:=true;
313: end;
314:
315: if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
316:     FSynSer.Config(BaudRateValue(), ConstsBits[FDataBits], ConstsParity[FParity],
317:         ConstsStopBits[FStopBits], FSoftflow, FHardflow);
318: end;
319: FFlowControl:=fc;
320: end;
321:
322: {
323: procedure TSdpoSerial.SetFlowControl(fc: TFlowControl);
324: begin
325:     if FHandle<>-1 then begin
326:         if fc=fcNone then CurTermIO.c_cflag:=CurTermIO.c_cflag and (not CRTSCTS)
327:         else CurTermIO.c_cflag:=CurTermIO.c_cflag or CRTSCTS;
328:         tcsetattr(FHandle,TCSADRAIN,CurTermIO);
329:     end;
330:     FFlowControl:=fc;
331: end;
332: }
333: procedure TSdpoSerial.SetParity(pr: TParity);
334: begin
335:     if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
336:         FSynSer.Config(BaudRateValue(), ConstsBits[FDataBits], ConstsParity[FParity],
337:             ConstsStopBits[FStopBits], FSoftflow, FHardflow);
338:     end;
339:     FParity:=pr;
340: end;
341:
342: procedure TSdpoSerial.SetStopBits(sb: TStopBits);
343: begin
344:     if FSynSer.Handle<>INVALID_HANDLE_VALUE then begin
345:         FSynSer.Config(BaudRateValue(), ConstsBits[FDataBits], ConstsParity[FParity],
346:             ConstsStopBits[FStopBits], FSoftflow, FHardflow);
347:     end;
348:     FStopBits:=sb;
349: end;
350:
351: function TSdpoSerial.WriteBuffer(var buf; size: integer): integer;
352: begin
353: // if FSynSer.Handle=INVALID_HANDLE_VALUE then
354: //     ComException('can not write to a closed port. ');
355:     result:= FSynSer.SendBuffer(Pointer(@buf), size);
356: end;
357:
358: function TSdpoSerial.WriteData(data: string): integer;
359: begin
360:     result:=length(data);
361:     FSynSer.SendString(data);
362: end;
363:
364:
365: function TSdpoSerial.ModemSignals: TModemSignals;
366: begin
367:     result:=[ ];
368:     if FSynSer.CTS then result := result + [ msCTS ];
369:     if FSynSer.carrier then result := result + [ msCD ];
370:     if FSynSer.ring then result := result + [ msRI ];
371:     if FSynSer.DSR then result := result + [ msDSR ];

```

```
372: end;
373:
374: function TSdpoSerial.GetDSR: boolean;
375: begin
376:     result := FSynSer.DSR;
377: end;
378:
379: function TSdpoSerial.GetCTS: boolean;
380: begin
381:     result := FSynSer.CTS;
382: end;
383:
384: function TSdpoSerial.GetRing: boolean;
385: begin
386:     result := FSynSer.ring;
387: end;
388:
389: function TSdpoSerial.GetCarrier: boolean;
390: begin
391:     result := FSynSer.carrier;
392: end;
393:
394: procedure TSdpoSerial.SetBreak(OnOff: boolean);
395: begin
396:     // if FHandle=-1 then
397:     //     ComException('can not set break state on a closed port. ');
398:     // if OnOff=false then ioctl(FHandle,TIOCCBRK,1)
399:     // else ioctl(FHandle,TIOCSBRK,0);
400: end;
401:
402:
403: procedure TSdpoSerial.SetDTR(OnOff: boolean);
404: begin
405:     FSynSer.DTR := OnOff;
406: end;
407:
408:
409: procedure TSdpoSerial.SetRTS(OnOff: boolean);
410: begin
411:     FSynSer.RTS := OnOff;
412: end;
413:
414:
415: procedure TSdpoSerial.ComException(str: string);
416: begin
417:     raise Exception.Create('ComPort error: '+str);
418: end;
419:
420: function TSdpoSerial.BaudRateValue: integer;
421: begin
422:     if FAltBaudRate > 0 then begin
423:         result := FAltBaudRate;
424:     end else begin
425:         result := ConstsBaud[FBaudRate];
426:     end;
427: end;
428:
429: { TComPortReadThread }
430:
431: procedure TComPortReadThread.CallEvent;
432: begin
433:     if Assigned(Owner.FOnRxData) then begin
```

```
434:      Owner.FOnRxData(Owner);
435:   end;
436: end;
437:
438: procedure TComPortReadThread.Execute;
439: begin
440:   try
441:     while not MustDie do begin
442:       if Owner.FSynSer.CanReadEx(100) then
443:         Synchronize(@CallEvent);
444:       end;
445:     finally
446:       Terminate;
447:     end;
448:
449: end;
450:
451:
452: procedure Register;
453: begin
454:   RegisterComponents('5dpo', [TSdpoSerial]);
455: end;
456:
457: initialization
458: {$i TSdpoSerial.lrs}
459:
460: end.
```