Univerza v Ljubljani

Fakulteta za elektrotehniko

Marko Ušaj

Implementacija generičnega ogrodja za komunikacijo med napravami tehničnega varovanja

Diplomsko delo univerzitetnega študija

Mentor: doc. dr. Boštjan Murovec

Ljubljana, september 2005

Zahvala

Zahvala gre mentorju doc. dr. Boštjanu Murovcu, čigar nasveti so to delo dali na plano. Prav tako se moram zahvaliti podjetju Robotina, d. o. o., Koper, posebej pa gospodu Juretu Hrvatiču za vso izkazano pomoč pri delu. Gospe Dragici Kocijančič se najlepše zahvaljujem za njeno dobro voljo pri lektoriranju diplomske naloge; brez nje bi prenekatera vejica potonila v morju besed.

Povzetek

Cilj diplomske naloge je izdelava generičnega gonilnika kot ogrodja za izvedbo poljubnega gonilnika za komunikacijo z napravami tehničnega varovanja. To ogrodje je del širšega programskega paketa za informatizacijo naprav tehnične zaščite, ki ga izdeluje slovensko podjetje.

V začetku dela je predstavljena problematika informatizacije naprav tehničnega varovanja. Kot ena izmed možnih rešitev za uspešno povezavo naprav različnih proizvajalcev, je predstavljen porazdeljen sistem za zajem podatkov s centraliziranim shranjevanjem. Na tej podlagi je izoblikovana groba struktura člena za zajem in lokalno obdelavo podatkov, katerega del je generični gonilnik. Določene so zahteve, ki jih mora tak člen izpolnjevati in pojasnjena vloga generičnega gonilnika.

Fina zasnova generičnega gonilnika je izdelana na podlagi konkretne implementacije. Najprej je podano ogrodje, abstraktni programski razred, ki določa metode, katere mora implementirati vsak gonilnik. Na tak način dobimo univerzalen vmesnik za upravljanje poljubnega gonilnika, vkolikor je izpeljan iz tega razreda. Iz te osnove so nato izpeljani še trije programski razredi, ki služijo kot pomožni razredi za izpeljavo različnih tipov gonilnikov, glede na uporabljeno komunikacijsko vodilo.

V nadaljevanju sta prikazani konkretni implementaciji dveh gonilnikov, ki sta izpeljana iz generičnega gonilnika. Prvi je gonilnik za Video Server. To je koncentrator za video snemalne naprave Geutebrueck Multiscope in omogoča prenos slike ter obveščanje o zaznanih premikih in izpadih video signala. Komunikacija poteka preko TCP/IP protokola. Drugi je gonilnik za Draeger Regard, sistem za merjenje koncentracije plinov. Ta nudi informacije o izmerjeni koncentraciji plinov, različnih stopnjah alarmov in morebitnih napakah v sistemu. Za komunikacijo je uporabljeno ProfiBus vodilo in protokol ProfiBus DP-V0.

Na koncu dela je primer delovanja celotnega sistema, kot je bil izveden v enem izmed večjih slovenskih podjetij.

Ključne besede: gonilniki, TCP/IP, ProfiBus, naprave tehničnega varovanja, objektno orientirano programiranje

Abstract

The focus of the thesis is to build generic device driver interface that would ease the driver development for communication with security devices regardless of the type and / or manufacturer. This interface is part of a software packet for interconnection of security devices produced by a Slovene company.

Beginning with the description of the typical communication problems that are likely to be expected in the field of security devices, a solution is proposed. This solution is based on distributed data acquisition points and centralised data storage. Using proposed solution, rough structure of the data acquisition and processing software part is made. In this context some basic demands are defined which must be full filed and the role of generic device driver is explained.

Fine structure of the generic device driver is made according to the real implementation. We define an abstract driver parent class containing common methods and properties that are inherited by all derived classes. This serves as the common interface to the actual device driver. Based on parent class three child classes are derived to ease the driver development. Each of them introduces additional methods and properties that simplify usage of different communication protocols.

Next, implementation of two real device drivers based on generic device driver class is shown. The first one is communicating with Integra Video Server, concentrator for Geutebrueck Multiscope digital video surveillance systems. Integra Video Server is capable of multiple simultaneous connections to Multiscope servers via TCP/IP protocol. It is used to relay various event messages (mostly motion detection events and video signal-related events) and stored images from connected Multiscope servers to Integra Video Clients over TCP/IP connection. The second driver shown is used to gather information from Draeger Regard gas detection systems. Here, Profibus connection is used, utilizing Profibus DP-V0 protocol. Regard system provides information about currently measured gas concentration and possible alarm or fault state.

Finally, an example of the real implementation of the entire system is shown as it was implemented in one of major Slovene company. This example shows how quite

complex systems can be made using distributed data acquisition method.

Key words: device drivers, TCP/IP, Profibus, Security Devices, object orientated programming

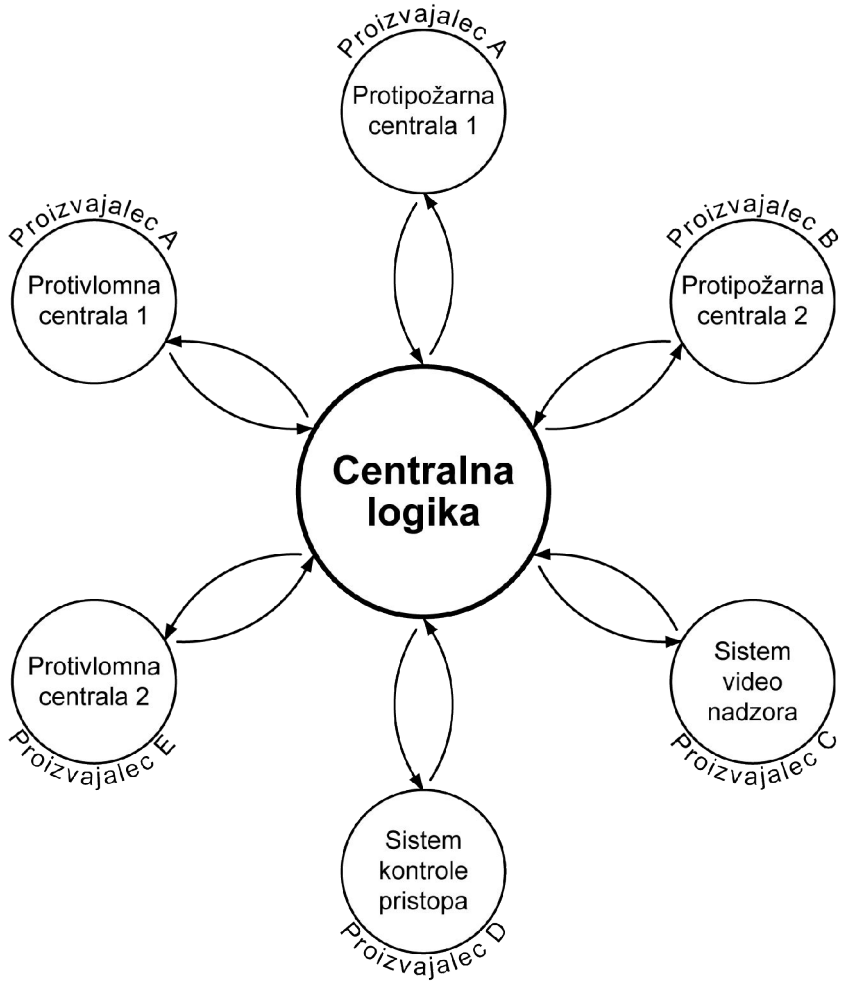
Vsebina

Seznam slik

Seznam programskih izsekov

1. Uvod

Dandanes se v sleherni industriji kakor tudi v objektih storitvene panoge, pojavljajo naprave tehničnega varovanja. Le-te vključujejo naprave za zaščito pred vlomom in/ali požarom (ta dva tipa sta najbolj pogosta), vedno bolj pa se uporablja še sisteme za video nadzor in sisteme za kontrolo pristopa. V različnih vrstah industrije lahko zasledimo še naprave za zaznavanje plinov, izvršne elemente zaščite (prezračevanje, šprinkler, inertni plini,...) in druge.



Slika 1: Shema splošne povezljivosti naprav tehničnega varovanja

Podjetja, ki to opremo izdelujejo, so ponavadi ozko specializirana za neko specifično področje. Tako na primer proizvajalec, ki izdeluje naprave za zaznavanje plinov, ne izdeluje protivlomnih central. Potemtakem bo kupec, ki želi uvesti celovit sistem

varovanja, imel ne samo pestre izbire izdelkov različnih konkurenčnih podjetij, pač pa tudi raznolike proizvajalce gradnikov postavljenega sistema (slika 1).

Druga značilnost proizvajalcev naprav tehničnega varovanja je njihova precejšnja konzervativnost pri povezovanju na druge naprave ne glede na namen povezovanja. Taka naravnanost je sicer smiselna z vidika zagotavljanja večje varnosti delovanja naprave, saj jo je tako težje onesposobiti, vendar je zato tudi težja interakcija z ostalimi napravami, ki bi pa lahko povečala varnostno učinkovitost celotnega sistema. Za primer navedimo situacijo, ko sta na nekem objektu nameščena sistema za protipožarno zaščito in kontrolo prehodov. Če ta dva sistema povežemo z neko logiko, lahko v primeru požara na objektu na prizadetem območju avtomatsko izdamo ukaz za brezpogojno odklepanje vseh vrat, ki so pod nadzorom sistema kontrole prehodov, in tako ljudem omogočimo lažje in varnejše umikanje iz območja ogroženosti.

1.1 Povezovanje naprav tehničnega varovanja

Danes so na trgu številni specializirani programski paketi za povezovanje naprav tehničnega varovanja, ki pa se osredotočajo predvsem na naprave določenega tipa ali določenega nabora proizvajalcev. Slednji imajo to prednost, da omogočajo boljšo povezljivost naprav, saj je navadno izbran tak nabor proizvajalcev, ki ima čimbolj združljivo obliko povezovanja. Med njimi so najbolj pogosti programski paketi določenega proizvajalca, ki omogočajo odlično povezljivost lastnega programa naprav, ne omogočajo pa povezovanja na naprave drugih proizvajalcev.

Druga možnost, ki jo lahko uporabimo za povezovanje naprav tehničnega varovanja, je uporaba splošnonamenskih SCADA[1] [1] programskih paketov. Primarni namen SCADA sistemov je vizualizacija proizvodnje in njenih ključnih parametrov na računalniških zaslonih, in s tem omogočiti jasno sliko trenutnega dogajanja v proizvodnem procesu ali delu proizvodnega procesa [2]. Ker lahko nekatere parametre iz naprav tehničnega varovanja predstavimo kot procesne spremenljivke (na primer zasičenost detektorja dima, zasičenost detektorja plina, odprtost vrat, prisotnost požarnega alarm, itd.), bi načeloma v ta namen lahko uporabili tak sistem. Vendar se tu pojavi problem predvsem glede velikega števila vhodnih točk, saj lahko

1 SCADA: Supervisory Control And Data Acquisition

denimo protipožarni sistem vključuje tudi do 1000 naslovljivih elementov oz. vhodnih točk, od katerih lahko vsak zajame vsaj dve stanji, običajno pa več. SCADA sistemi, ki dovoljujejo veliko število vhodnih točk, so navadno strojno in cenovno potratni.

Drug problem, ki omejuje uporabnost SCADA sistemov za povezovanje naprav tehničnega varovanja je ta, da gonilniki za večino teh naprav ne obstajajo. Ker večina SCADA sistemov uporablja OPC tehnologijo[2] [3] za dostop do posameznih naprav, je ta problem načeloma rešljiv z implementacijo ustreznih OPC strežnikov, vendar je taka rešitev zelo kompleksna in podraža tako izvedbo kot vzdrževanje sistema.

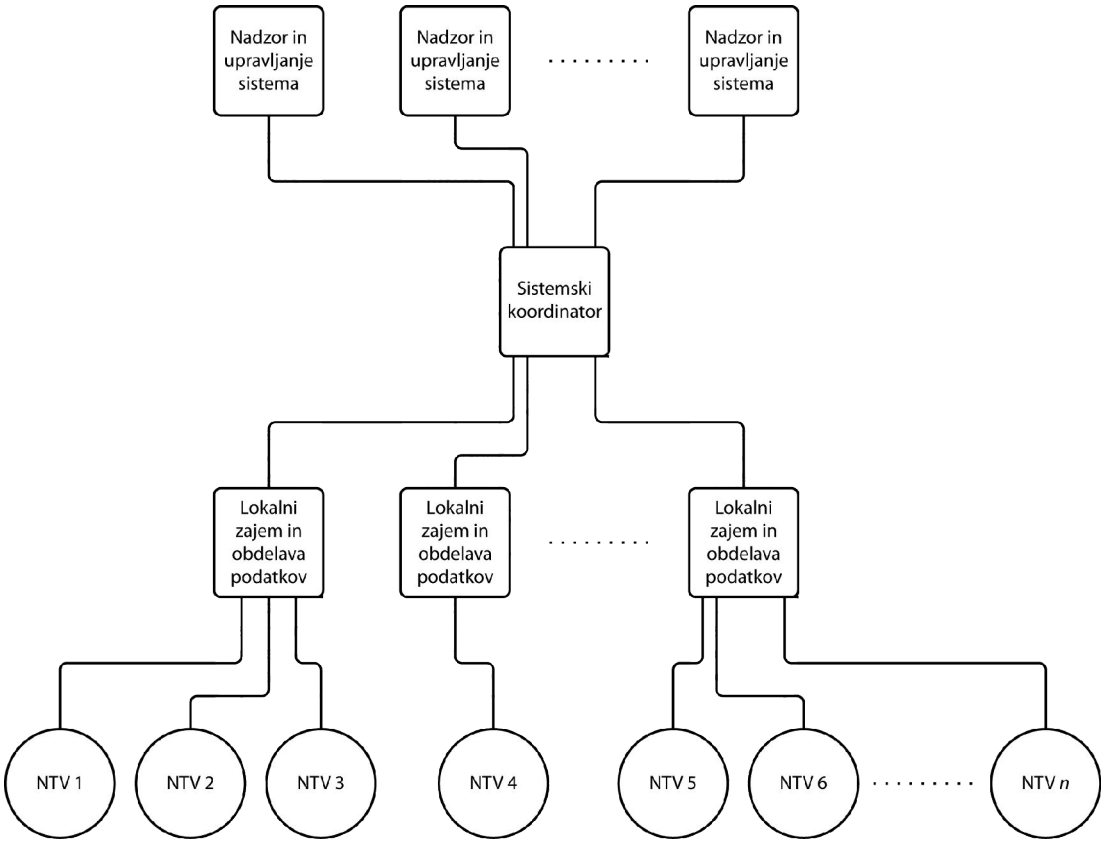
2 OPC: OLE for Process Control

2. Programski okvir komunikacijskih vmesnikov

Če želimo postaviti zasnovo komunikacijskih vmesnikov, moramo najprej določiti izhodišča, na katerih jo gradimo.

Naravnanost nove generacije industrijskih naprav je, da se obdelava podatkov vrši čim bližje zajemnemu mestu. V primeru naprav, ki so obravnavane v tem delu, to pomeni, da bi se morala obdelava vršiti v tistem logičnem delu, ki z napravami komunicira. Poleg tega je potrebno upoštevati še dejstvo, da komunikacija s posamezno napravo porabi del procesne moči računalnika, na katerem le-ta poteka. Jasno je torej, da obstaja neka zgornja meja števila naprav, ki jih lahko priključimo na en računalnik tako, da zadostimo pogoju pričakovane odzivnosti komunikacije.

Na podlagi naštetih dejstev je bila osnovana arhitektura porazdeljenega sistema zajema podatkov, kot je prikazana na sliki 2.



Slika 2: Groba zasnova distribuiranega sistema zajema podatkov

Zajem in obdelava podatkov sta zasnovana tako, da se odvijata lokalno, čim bližje sami napravi (na sliki 2 označena kot NTV – naprava tehničnega varovanja). Ta zasnova se izkaže za nadvse primerno, če upoštevamo, da je večina komunikacijskih vmesnikov na napravah tehničnega varovanja serijskih, in sicer tipa RS232 [4]. Komunikacijski vmesnik, ki deluje po tem standardu, ima najdaljšo razdaljo prenosa nominalno 15 metrov[3], glede na okoliščine uporabe pa se ta lahko znatno skrajša [5]. Na predlagan način lahko računalnik, na katerem deluje komunikacijski program, pomaknemo v neposredno bližino zajemnega mesta (npr. protipožarne centrale).

Kot osrednji člen je postavljen sistemski koordinator. To je program, ki povezuje posamezne porazdeljene člene sistema. Njegova naloga je, da lokalno zajete podatke sinhronizira in posreduje na nadzorni in upravljalski nivo ter izvrši morebitne avtomatizme, ki se jih določi v sistemu. Prav tako je v tem členu izveden mehanizem hrambe podatkov in nadzora nad posameznimi členi sistema.

Za vizualizacijo stanja sistema skrbi tretji člen, preko katerega lahko tudi upravljamo s sistemom.

Eden od pomembnejših dejavnikov pri načrtovanju sistema je izbira primernega medija za prenos podatkov med posameznimi členi sistema. Zaradi široke uporabe, nizke cene in velike fleksibilnosti smo za izvedbo uporabili protokol TCP/IP[4] [6].

Pričujoče delo se osredotoča na člen, ki skrbi za lokalni zajem in obdelavo podatkov.

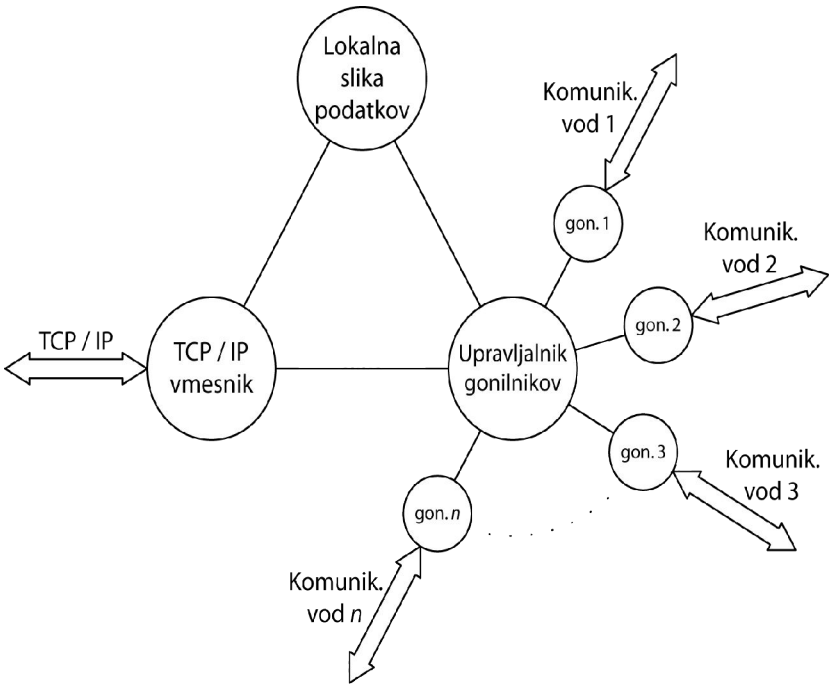
3 Omejitev pravzaprav izhaja iz dejstva, da standard določa maksimalno kapacitivnost vodnika 2500 pF/m pri hitrosti prenosa podatkov 19200 baudov. Podana razdalja je tako navedena v primeru vodnika s kapacitivnostjo 50 pF/m in se navadno uporablja za izhodišče. V kolikor prepolovimo hitrost prenosa podatkov, se dolžina poveča za faktor 10.

4 TCP – Transfer Control Protocol, IP – Internet Protocol

2.1 Groba zasnova

Za določitev grobe zasnove člena, ki skrbi za lokalni zajem podatkov (v nadaljevanju *komunikacijski člen*), je potrebno določiti osnovne naloge, ki jih naj izvršuje. Te naloge naj bodo naslednje:

- mrežno upravljanje: komunikacijski člen mora imeti možnost daljinskega upravljanja, parametriranja in nadziranja preko mrežnih povezav;
- podatkovna neodvisnost: komunikacijski člen mora imeti lastno sliko zadnjih veljavnih podatkovnih struktur sistema, tako da je sposoben neodvisnega delovanja v primeru izpada mrežne povezave na centralni člen;
- komunikacija z napravami: komunikacijski člen mora znati komunicirati na priključene naprave tehničnega varovanja;
- predpriprava podatkov: da se razbremeni centralni člen, mora komunikacijski člen opraviti nalogo filtracije pridobljenih podatkov tako, da se na centralni nivo posreduje samo koristne podatke[5].



Slika 3: Groba zasnova komunikacijskega člena

Grafična ponazoritev teh zahtev je na sliki 3. Za mrežno upravljanje skrbi TCP/IP

5 Koristni podatki v tem primeru pomenijo tiste podatke, ki jih uporabnik v sistemu definira kot opazovane in so odvisni od gonilnika za posamezno napravo.

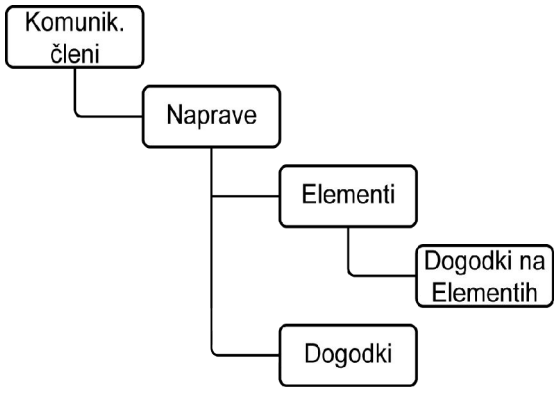
vmesnik, podatkovno neodvisnost zagotavlja lokalna slika podatkov, komunikacija z napravami se vrši preko gonilnikov (na sliki označeno kot *gon. x*), za predpripravo podatkov pa poskrbi upravljalnik gonilnikov.

2.1.1 Lokalna slika podatkov

Lokalna slika podatkov je eden pomembnejših gradnikov sistema. Nudi hitrejši dostop do podatkov, ki so pomembni za posamezen komunikacijski člen, in skrbi za nemoteno delovanje gonilnikov med morebitnim izpadom komunikacije.

Sinhronizacijo podatkov preko celotnega sistema je potrebno zagotavljati preko mrežnih povezav in neodvisno od ostalih gradnikov komunikacijskega člena. Podatki v lokalni sliki morajo biti logično urejeni. Nudi se jih ostalim členom kot ustrezen programski objekt. Njegove lastnosti nudijo dostop do informacij relevantnih za druge funkcije komunikacijskega člena. Vsak tak objekt je lahko tudi nosilec drugih objektov, v kolikor je informacijo, ki jo nosi, moč razčleniti na logične podsklope. Na tak način lahko dosežemo transparentno delovanje po načelu objektno orientiranega programiranja in zagotovimo pregledno strukturo.

Struktura podatkov in njihova hierarhija morata biti skupni celotnemu sistemu. Če tega ne zagotovimo, se lahko zgodi, da bo nek člen sistema določen podatek drugače interpretiral kot preostali členi, kar neizbežno vodi v napačno delovanje sistema.



Slika 4: Groba podatkovna struktura

Na sliki 4 je mogoče videti grobo podatkovno strukturo, ki ji pričujoče delo sledi. Vsakemu komunikacijskemu členu iz neke množice priredimo določeno število naprav. Napravi je prirejena množica elementov in množica dogodkov. Vsak element

ima določeno množico dogodkov na elementih, ki je podmnožica množice dogodkov na napravi.

2.1.2 TCP/IP vmesnik

Komunikacije preko danes obstoječih omrežij v veliki meri potekajo preko standardnih vmesnikov, med katerimi je verjetno najbolj razširjen TCP/IP komunikacijski protokol. Ta je bil prvotno namenjen povezavi oddaljenih računalnikov, vendar se je danes razširil tudi na druga področja, kot so denimo sistemi procesnega vodenja, inteligentne zgradbe, logistika, itd. Prevladuje v omrežjih tipa *Ethernet*, uporablja pa se ga lahko tudi pri *USB* [7] in *FireWire* [8] vodilih ter v industriji pogosto uporabljenih fieldbus vodilih (naprimer *Profibus* in *Modbus*) [9].

Uporabnost TCP/IP protokola je praktično neomejena. Elektronska pošta, prenos video signala, internetne strani, RS232 in RS485 pretvorbe, prenos DNC[6] datotek na CNC[7] stroje in povezovanje v CAD/CAM[8] sistemih [10] so samo nekatere implementacije, ki kot nosilec uporabljajo ta protokol.

Odločitev o uporabi TCP/IP protokola je osnovana tudi na dejstvu, da je ta protokol dobro preizkušen in zanesljiv. Morebitne težave se lahko hitro in sistematično ugotovi ter odpravi.

Naloga TCP/IP vmesnika komunikacijskega člena je ta, da vzdržuje povezavo s centralnim členom in pretvarja prejete podatke v klice funkcij s pripadajočimi parametri. Prav tako skrbi za vrnitev podatkov centralnemu členu. Čeprav nam TCP/IP protokol zagotavlja integriteto podatkov, pa se v praksi izkaže, da v bolj problematičnih okoliščinah (preobremenjenost računalniškega omrežja, elektromagnetne motnje, zastarela infrastruktura) pride do sprejema napačnih podatkov. Zaradi tega je priporočljivo v protokol komunikacije vključiti preverjanje integritete podatkov, na primer CRC32 ali MD5.

V primeru povezave preko javnih računalniških omrežij je nujno uvesti še šifriranje

6 DNC: Direct Numerical Control – neposredno numerično krmiljenje
7 CNC: Computer Numerical Control – računalniško numerično krmiljenje
8 CAD/CAM: Computer Aided Design / Computer Aided Manufacturing – računalniško podprto načrtovanje / računalniško podprta proizvodnja

prenosa podatkov. Le-to mora biti izvedeno tako, da se podatke dešifrira takoj po sprejemu oziroma tik pred oddajo, tako da drugi deli programa delujejo neodvisno od tega, ali je šifriranje povezave izvedeno ali ne [11].

2.1.3 Upravljalnik gonilnikov

Upravljalnik gonilnikov je osrednji gradnik komunikacijskega člena. Glede na nastavitve, ki so pridobljene preko lokalne slike podatkov, se naredijo programski objekti gonilnikov, preko katerih je nato izvedena komunikacija z napravami tehničnega varovanja. Upravljalnik gonilnikov skrbi za pravilno inicializacijo posameznih gonilnikov, za njihovo finalizacijo (programski izklop), ustavljanje in zagon gonilnikov, morebitno resetiranje ob novih podatkih, zagotavlja nadzor nad delovanjem gonilnikov (t.i. *"časovni čuvaj"* ali angleško *"watchdog timer"* [9]) ter skrbi za prenos podatkov iz gonilnika do TCP/IP vmesnika, od koder se nato posredujejo na centralni člen.

2.1.4 Gonilnik

Osnovna naloga gonilnika je vzpostavitev in vzdrževanje komunikacije z napravo. Ta komunikacija mora omogočati vpogled v notranja stanja naprave, iz katerih lahko nato sklepamo, kaj se v sistemu dogaja. Zaželjeno je, da se na napravo lahko pošlje določene zahteve, na katere ta ustrezno reagira.

Na tem mestu je pomembno poudariti, da tukaj omenjeni gonilniki niso sistemski (ang. *Kernel Device Drivers*) [12], temveč so vezani na določen program, ki se v sistemu izvaja (programski gonilniki – ang. *Application Device Drivers*). Morda bistvena razlika med tema dvema tipoma gonilnikov je ta, da so sistemski gonilniki na voljo vsem programom, ki delujejo v določenem operacijskem sistemu in imajo standardizirane vmesnike, medtem ko so programski gonilniki pisani namensko za določeno aplikacijo, njihovi vmesniki pa niso nujno standardizirani. Lahko pa slednji uporabljajo sistemske gonilnike, navadno za dostop do strojne opreme.

9 časovni čuvaj: tehnika nadzora izvajanja programske kode. Nadzorna nit v enakomernih časovnih intervalih povečuje binarni števec, katerega nadzirani nit občasno postavi na 0. V kolikor vrednost binarnega števca doseže določeno vrednost, nadzorna nit nadzirano ponovno zažene.

3. Fina zasnova

Fina zasnova bazira na konkretni realizaciji člena, opisanega v prejšnjem poglavju. Člen zajema in lokalne obdelave podatkov je del programskega paketa *Integra Security*, ki je bil razvit na podjetju Robotina, d. o. o., Koper in predstavlja celovito rešitev informatizacije naprav tehničnega varovanja. Komercialno ime obravnavanega člena je *Remote Communicator* (oddaljeni komunikator) ali krajše *RemComm*. Na sliki 5 je prikazan uporabniški vmesnik oddaljenega komunikatorja.

Slika 5: Uporabniški vmesnik oddaljenega komunikatorja

Paket Integra Security danes uporablja večje število domačih in tujih podjetij ter državnih ustanov za centraliziran nadzor nad sistemi tehničnega varovanja. Med uporabniki so Gorenje, d. d., Velenje, Krka, d. d., Novo mesto, Tobačna Ljubljana, d. d., Ljubljana, Ljubljanska banka, d. d., Ljubljana ter druge.

Deluje v Microsoft Windows operacijskih sistemih NT 4, 2000 in XP. Za svoje delovanje potrebuje pravilno nameščen mrežni vmesnik in nameščeno podporo za TCP/IP protokol. Tip mrežnega vmesnika ni važen, pomembno je le, da omogoča prenos podatkov preko omenjenega protokola.

3.1 Uporabljeni termini

Struktura oddaljenega komunikatorja je zasnovana na programskih objektih in objektno orientiranem programiranju. Programski jezik je *Object Pascal* [13], uporabljeno razvojno okolje pa *Borland Delphi 7*.

Uporabljena nomenklatura in zapisi izhajajo iz ustaljenega angleškega in mednarodnega strokovnega izrazoslovja in dogovorov. V nadaljevanju so predstavljeni tisti termini, ki so v delu uporabljeni:

- funkcija, procedura: velikokrat se uporabljata izraza rutina ali metoda, t.j. zaporedje programskih stavkov, ki jih lahko kličemo iz različnih delov programa. Funkcije so metode, ki vračajo rezultat, procedure pa metode, ki ne vračajo rezultata;
- programski razred, razred: je programska struktura, sestavljena iz polj, metod in lastnosti.
  Polja so interne spremenljivke nekega razreda, metode pa procedure in funkcije, definirane znotraj razreda.
  Lastnosti so tista polja, ki jim določimo način dostopa. Dostop je lahko dovoljen samo za branje, samo za pisanje ali branje in pisanje. Za branje in/ali pisanje lastnosti lahko priredimo neko metodo, ki poskrbi za upravljanje z vrednostjo lastnosti in morebiti nanjo povezanimi strukturami;
- programski dogodki: programske lastnosti razreda, ki shranjujejo referenco na metodo nekega objekta. Navadno se jih uporablja za določevanje metod, ki naj se jih kliče ob izvrševanju določenega programskega sklopa v metodah razreda;
- objekt: dinamično zavzet del pomnilnika, njegovo strukturo narekuje razred, kateremu objekt pripada. Vsak objekt ima lastno kopijo polj razreda, vendar si pa vsi objekti istega razreda delijo iste metode. Objekti se naredijo in uničijo z uporabo posebnih metod, imenovanih konstruktorji in destruktorji;
- instanca: dinamično zavzet pomnilnik. Instanci navadno pripada tudi tip, to je opis strukturiranja pomnilnika. Velikokrat se za objekt uporablja tudi izraz instanca razreda;
- implementacija: programski stavki, ki realizirajo pričakovano obnašanje;

Izseki programske kode so zapisani v enakomerno široki pisavi in uokvirjeni. Na koncu izseka je njegov kratek opis z zaporedno številko, kot je razvidno iz naslednjega primera:

```
1. programski stavek
2. programski stavek
...
n. programski stavek
```

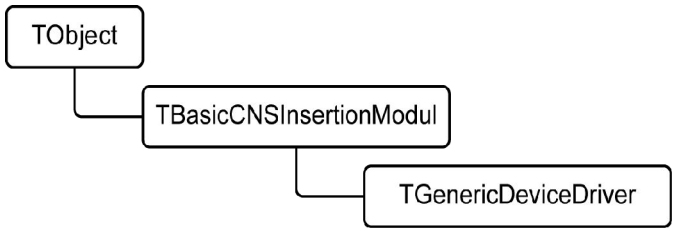
Izsek X: Primer prikaza programskega izseka

Navedbe delov programske kode v besedilu so ravno tako zapisane v enakomerno široki pisavi (primer: `TPrimerRazreda`).

Po priporočilih programskega jezika Object Pascal se imena razredov začnejo z veliko črko T (naprimer: `TPoljubenRazred`). Imena polj razreda se začnejo z malo črko f (naprimer: `fSeznam`). Izjemoma je za kritične odseke uporabljena začetnica `cs` tam, kjer je drugače težko razločiti med imeni polj.

3.2 Splošni gonilnik

Splošni gonilnik je osnovni programski razred za izpeljavo poljubnega gonilnika. Predstavi tiste metode, ki so skupne vsem gonilnikom, in tako omogoča univerzalen dostop do kateregakoli gonilnika.



Slika 6: Hierarhija razreda TGenericDeviceDriver

Razred je imenovan `TGenericDeviceDriver` in je izpeljan iz razreda `TBasicCNSInsertionModul` (slika 6). Definicija razreda je razvidna iz programskega izseka 1.

```
  TGenericDeviceDriver = class(TBasicCNSInsertionModul)
  private
    fOnError: TDriverMessage;
    fOnStatusChange: TDriverStatusChange;

    csDebugWindowSync: TCriticalSection;

    fDebugWindow: TForm;
    fOnDriverInfo: TDriverMessage;
```

```
    fOnEvent: TDriverEvent;
    fHoldCnt: integer;
    fFirstCycle: boolean;

    function GetDebugWindowVisible: boolean;
    procedure SetDebugWindowVisible(const Value: boolean);

    procedure SetHold(const Value: boolean);
    function GetHold: boolean;
  protected
    fStatus: TRemoteDeviceStats;
    fLink: TRemDev;
    fCycleThread: TCycleThread;
    csThreadRun: TCriticalSection;

    procedure Cycle; virtual;

    procedure NotifyError(Msg: string);
    procedure NotifyEvent(Event: THoldEvent);
    procedure NotifyStatChange(newStat: TRemoteDeviceStats);
    procedure NotifyInfo(const Info, SubInfo: string); override;

    procedure CreateEvent(const Adr: string; const EventData:
string;
      const DT: TDateTime = 0; const Additional: string = '';
      const Uporabnik: string = ''; const Input: string = '');

    function GetDriverVersion: TModulVersion; virtual; abstract;
    function GetDriverInfo: string; virtual; abstract;

    property FirstCycle: boolean read fFirstCycle write fFirstCycle;
  public
    constructor Create; reintroduce;
    destructor Destroy; override;

    procedure Initialize(_Link: TRemDev); virtual;
    procedure Finalize; virtual;

    procedure Run; virtual;
    function Stop(const timeout: integer = 5000): boolean; virtual;

    procedure CancelHold;

    procedure ExecuteCommand(const Command: string); virtual;
    procedure ExecuteTestEvent(const Adr: string; const EventData:
string);
      virtual;

    property Link: TRemDev read fLink;
    property Status: TRemoteDeviceStats read fStatus;

    property DriverVersion: TModulVersion read GetDriverVersion;
    property DriverInfo: string read GetDriverInfo;

    property DebugWindowVisible: boolean read GetDebugWindowVisible
      write SetDebugWindowVisible;

    property Hold: boolean read GetHold write SetHold;
    property OnError: TDriverMessage read fOnError write fOnError;
    property OnEvent: TDriverEvent read fOnEvent write fOnEvent;
    property OnStatusChange: TDriverStatusChange read
```

```
fOnStatusChange
      write fOnStatusChange;
    property OnDriverInfo: TDriverMessage read fOnDriverInfo
      write fOnDriverInfo;
  end;
```

Izsek 1: Definicija objekta TGenericDeviceDriver

`Create` in `Destroy` sta konstruktor in destruktor objekta. Metoda `Create` se kliče ob kreiranju objekta za tem, ko je zanj zavzet prostor v pomnilniku, torej za implementacije metod (funkcij in procedur) in alokacije internih spremenljivk . V njej je izvedena inicilizacija internih spremenljivk na začetne vrednosti, inicializacija kritičnih odsekov (`csThreadRun`, `csDebugWindowSync`) in kreiranje programskih niti, v katerih nato gonilnik deluje (`fCycleThread`). Direktiva `reintroduce` določa, da smo prejšnjo deklaracijo metode namenoma nadomestili z novo, tako da nam prevajalnik ne javlja namiga o morebitni napaki.

Metoda `Destroy` se kliče ob sprostitvi objekta, preden se sprosti pomnilnik, rezerviran za objekt. V njej se izvede finalizacija objekta in sprostijo vsi kritični odseki in programske niti.

`Initilize` metodo se kliče takrat, ko so na voljo podatki o gonilniku. Parameter `_Link` je kazalec, ki kaže na objekt tipa `TRemDev`. V njem so logično strukturirani podatki, ki se nanašajo na določen gonilnik, kot je na primer inicializacijski niz, naslovi elementov in dogodki naprave, ime in tip naprave, na katero komuniciramo, itd. Direktiva `virtual` naznani, da se bo implementacija metode v kasnejših izpeljavah objekta spreminjala.

`Finalize` metoda je namenjena pravilni sprostitvi pomnilnika, zaseženega med delovanjem gonilnika, pred tem pa kliče metodo za zaustavitev komunikacije z napravo.

`Run` in `Stop` metodi sta namenjeni zagonu in zaustavitvi programske niti, ki periodično kliče metodo `Cycle` (glej poglavje 3.4.3 Delovanje gonilnika). Kličemo ju lahko šele po tem, ko je objekt bil pravilno iniciliziran preko metode `Initilize`. Ker obstaja možnost, da se metodi kliče hkrati (delovanje gonilnika je namreč možno nadzorovati lokalno in daljinsko), je njuno delovanje sinhronizirano preko kritičnega odseka `csThreadRun`. Metodi `Stop` lahko podamo parameter, ki

določa, koliko milisekund ima gonilnik na voljo, da ustavi delovanje, drugače metoda vrne vrednost `false`. Privzeta vrednost je 5000 ms.

Metoda `CancelHold` se uporablja takrat, ko je gonilnik v stanju zadrževanja podatkov in želimo to stanje prekinit (glej poglavje 3.4.3 Delovanje gonilnika).

`ExecuteCommand` metoda je namenjena pošiljanju ukazov na priključeno napravo. Ukazi se lahko generirajo na zahtevo uporabnika ali kot posledica nekega dogodka v sistemu. Posredujejo se v obliki niza znakov in so odvisni od izvedbe gonilnika.

`ExecuteTestEvent` metoda proži interni test sporočanja dogodkov iz gonilnika. Dogodki, ki nastanejo kot posledica te metode, so označeni kot testni in se v sistemu ne shranjujejo.

Lastnost objekta `Link` kaže na podatkovno strukturo, ki je bila podana gonilniku ob inicilizaciji in služi za enolično povezavo med gonilnikom in podatkovno strukturo.

Lastnost `Status` določa trenutno stanje gonilnika. Možna stanja so:

- `rdsDisabled`: objekt gonilnika ni naložen,
- `rdsInit`: komunikacija z napravo se inicializira,
- `rdsRun`: komunikacija z napravo deluje normalno,
- `rdsStop`: komunikacija z napravo je bila ustavljena,
- `rdsHold`: komunikacija deluje, vendar je gonilnik v stanju zadrževanja dogodkov,
- `rdsError`: napaka na komunikaciji, komunikacija ne deluje,
- `rdsFatalError`: kritična napaka delovanja gonilnika.

Lastnost `DriverVersion` oziroma metoda `GetDriverVersion` vrača različico gonilnika. Metoda je določena kot abstraktna, saj jo mora implementirati vsak posamezen gonilnik in ne more biti skupna za vse gonilnike. Podobno velja za lastnost `DriverInfo` oziroma metodo `GetDriverInfo`, ki vrača informativni opis gonilnika, na primer izdelovalca, model naprave, leto izdelave, itd.

Preko lastnosti `DebugWindowVisible` lahko določamo vidnost okna, skozi katerega imamo vpogled v komunikacijo med napravo in gonilnikom v živo.

Lastnost `Hold` nosi informacijo, ali je gonilnik v stanju zadrževanja dogodkov. Prav

tako lahko gonilnik preko te lastnosti postavimo v stanje zadrževanja.

Dogodek `OnError` se proži takrat, ko se na gonilniku pojavi kritična napaka (stanje `rdsFatalError`). Preko dogodka `OnEvent` se javlja dogodke na napravi, zapisane v začasni, podatkovno neodvisni strukturi (`THoldEvent`). Dogodek `OnStatusChange` bo prožen takrat, ko se spremeni stanje gonilnika (lastnost `Status`), dogodek `OnDriverInfo` pa v primeru, da gonilnik javi informativni niz, ki se navadno uporablja za opazovanje komunikacije.

Metoda `Cycle` se ciklično kliče iz programske niti, ki pripada določenemu gonilniku. Namen metode je, da se poenostavi pisanje gonilnikov tako, da je njihovo izvajanje podobno izvajanju PLK[10] programov. Tako naj implementacija vsebuje proceduralno zapisan program za vzpostavitev in vzdrževanje komunikacije s priključeno napravo. Ob vsakem ciklu (torej klicanju metode `Cycle`) naj se preveri, ali je v pomnilniku komunikacijskega vmesnika na voljo nov podatek, nakar se ga prebere in ustrezno obdela. Čakanje na sprejem celotnega paketa podatkov znotraj metode se odsvetuje, saj se lahko dogodi, da program zaide v neskončno zanko.

Metoda `CreateEvent` poenostavi generiranje dogodkov v gonilniku in vsebuje vse mehanizme za posredovanje nastalih dogodkov naprej. Parametra, ki določata naslov (`Adr`) in podatek o tipu dogodka (`EventData`), sta obvezna, preostali parametri imajo privzete vrednosti:

- `DT`: če je vrednost parametra 0, se dogodku vpiše sistemski čas nastanka (privzeto), drugače vrednost določa čas nastanka;
- `Additional`: dodatni opis dogodka, ki ga gonilnik lahko sporoči in je spremenljive narave;
- `Uporabnik`: parameter se uporablja pri sistemih kontrole pristopa in nosi informacijo o številki kartice in / ali imenu in priimku osebe, zaradi katere je dogodek nastal;
- `Input`: parameter določa trenutno vrednost neke spremenljive veličine iz priključene naprave, ki jo lahko predstavimo na elementu (na primer koncentracija plina, ki jo javlja merilna glava, zasičenost optičnega javljalnika dima, itd.). Pri tem tipa dogodka ni nujno potrebno podajati in je

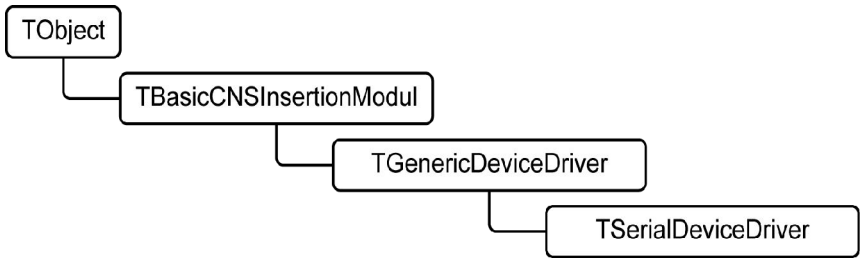
10 PLK: programabilni logični krmilnik

dovoljeno uporabiti prazen niz (`EventData = ''`), saj se lahko vrednost spremeni, ne da bi pri tem sprožila nek dogodek.

Lastnost `FirstScan` pove gonilniku, če je trenutni klic metode `Cycle` prvi ali ne. To se navadno uporablja za inicializacijo komunikacije z napravo znotraj metode `Cycle`.

3.2.1 Objekt serijskega gonilnika

Objekt serijskega gonilnika, imenovan `TSerialDeviceDriver`, je podlaga za vse gonilnike, ki delujejo preko standardne serijske povezave RS-232. Predstavi metode za upravljanje s serijskimi (COM) vrati, metode za branje in pisanje podatkov ter poenostavlja inicializacijo gonilnika.



Slika 7: Hierarhija razreda TSerialDeviceDriver

Starševski razred je `TGenericDeviceDriver` (slika 7), iz katerega izpeljemo metode, ki so skupne vsem gonilnikom, in predstavimo nove (glej izsek 2).

```
TSerialDeviceDriver = class(TGenericDeviceDriver)
private
  FDsrFlow: boolean;
  FCtsFlow: boolean;
  FDsrSens: boolean;
  fHandle: Cardinal;
  fBaud: integer;
  fPort: integer;
  fBitNo: TBitNo;
  FDtrFlow: TDtrControl;
  fParity: TParity;
  FRtsControl: TRtsControl;
  fStopBitNo: TStopBitNo;
protected
  function Open: Boolean;
  procedure Reset;
  procedure Close;
  function IsOpen: boolean;
  function TestByte: boolean;
  function GetByte: Byte;
  procedure WritePort(const sData: string);
  function ReadPort: string;
```

```
    function GetInCount: integer;

    property Handle: Cardinal read fHandle;
    property Parity: TParity read fParity write fParity;
    property BitNo: TBitNo read fBitNo write fBitNo;
    property StopBitNo: TStopBitNo read fStopBitNo write fStopBitNo;
    property Baud: integer read fBaud write fBaud;
    property Port: integer read fPort write fPort;
    property CtsFlowControl: boolean read FCtsFlow write FCtsFlow;
    property DsrFlowControl: boolean read FDsrFlow write FDsrFlow;
    property DtrFlowControl: TDtrControl read FDtrFlow
      write FDtrFlow;
    property DsrSensitivity: boolean read FDsrSens write FDsrSens;
    property RtsControl: TRtsControl read FRtsControl
      write FRtsControl;
  public
    procedure Initialize(_Link: TRemDev); override;
    procedure Finalize; override;
    function Stop(const timeout: Integer = 5000): Boolean; override;
  end;
```

Izsek 2: Definicija objekta TSerialDeviceDriver

Metoda `Initilize` je implementirana tako, da iz inicializacijskega niza, ki ga dobimo preko parametra `_Link`, razbere nastavitve serijskih vrat in temu primerno nastavi interne spremenljivke.

Metoda `Stop` privzame obnašanje svoje predhodnice ter poskrbi za pravilno zapiranje serijskih vrat.

Metoda `Open` odpre serijska vrata glede na nastavitve določene preko ustreznih lastnosti objekta. Odpiranje vrat je izvedeno preko Win32 API [14] funkcije `CreateFile` in omogoča neposredno naslavljanje 255 COM vrat. Dostop je polni (branje in pisanje), binarni. Nastavitev vrat je izvedena preko API funkcije `SetCommState`, kjer podamo željene nastavitve v `DCB`[11] strukturi.

`Reset` metoda kliče API funkcijo `PurgeComm`, ki sprosti vse morebitne podatke, ki čakajo v pomnilniku serijskega vmesnika operacijskega sistema.

`Close` metoda zapre vrata tako, da kliče API funkcijo `CloseHandle`. To metodo je potrebno izvesti pri ustavljanju gonilnika, za kar avtomatsko poskrbi metoda `Stop`.

Metoda `IsOpen` vrne vrednost `true`, če so vrata trenutno odprta, in vrednost `false`, če niso.

11 DCB: Device Control Block – kontrolni blok naprave; podatkovna struktura, ki določa nastavitvene parametre računalniške opreme.

`TestByte` kliče API funkcijo `ClearCommError`, od koder ugotovi, če so v pomnilniku serijskih vrat kakšni podatki, ki čakajo na sprejem. V tem primeru vrnejo vrednost `True`, `False` pa če neprebranih podatkov ni. Ta metoda je navadno predpogoj za branje oziroma sprejem podatkov.

`GetByte` prebere en zlog iz pomnilnika serijskih vrat s klicem API funkcije `ReadFile` in ga vrne v binarni obliki.

`WritePort` metoda je namenjena pošiljanju niza znakov preko odprtih serijskih vrat. Za pošiljanje uporablja API funkcijo `WriteFile`. Dolžine podatkov metoda ne omejuje, tako da je odločitev o dolžini pošiljanih podatkov v domeni klicatelja.

`ReadPort` metoda prebere vse, kar se trenutno nahaja v pomnilniku serijskih vrat in prebrano vrne klicatelju kot niz znakov. Za branje je uporabljena API funkcija `ReadFile`. Ker so pomnilniki serijskih vrat navadno manjših kapacitet (reda nekaj sto bajtov do enega kilobajta), ni bojazni, da bi sistem lahko na tak način padel v zasičenje.

Metoda `GetInCount` vrne število zlogov, čakajočih na sprejem. Uporablja enako API funkcijo kot `TestByte` metoda, le da vrača dejansko število zlogov, ki se nahajajo v pomnilniku serijskega vmesnika.

Lastnost `Handle` omogoča dostop do ročice (ang. *handle*) serijskih vrat, ki jo moramo podajati pri klicih API funkcij, da določimo, na katera vrata naj se klic funkcije nanaša. Ročico določi klic metode `Open` in nosi vrednost `INVALID_HANDLE_VALUE` (določeno kot Win32 konstanta), v kolikor vrata niso pravilno odprta.

Lastnost `Parity` določa, kakšen način preverjanja parnosti bomo uporabili. Možne vrednosti so:

- `parNO`: brez parnostnega bita,
- `parEVEN`: soda parnost,
- `parMARK`: parnostni bit vedno na logičnem visokem nivoju (1),
- `parODD`: liha parnost.

Preko lastnosti `BitNo` določimo, kolikšno je število podatkovnih bitov, ki se

prenašajo po serijskem kanalu. Možni sta vrednosti `bn8` ali `bn7`, kjer prva pomeni 8 podatkovnih bitov, druga pa 7.

`StopBitNo` lastnost se uporablja za določitev števila končnih bitov[12]. Zavzame lahko eno od naslednjih vrednosti:

- `sbnONE`: uporabljen je en končni bit,
- `sbnONEANDHALF`: uporabljen je en končni bit in pol,
- `sbnTWO`: uporabljena sta dva končna bita.

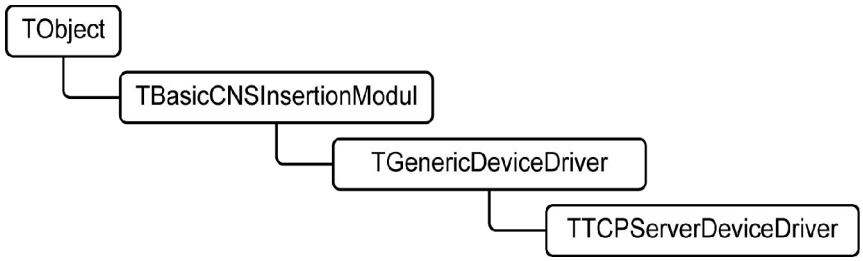
Lastnost `Baud` določa hitrost prenosa podatkov preko serijskih vrat in lahko zavzame poljubno vrednost. Navadno pa komunikacija poteka preko ene od standardnih hitrosti (podano v baudih): 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 56000, 57600, 115200, 128000, 256000. Hitrost podatkov, podana v baudih, pomeni dejansko hitrost podatkovnega voda in ne hitrosti prenosa koristnih podatkov. Hitrost prenosa koristnih podatkov je nekoliko nižja in se podaja v bitih ali bajtih na sekundo, odvisna pa je od nastavitev komunikacije (na primer od števila končnih bitov, števila podatkovnih bitov, parnosti, itd.).

Lastnost `Port` je namenjena določitvi številke serijskih vrat (COM1, COM2, ...), preko katerih se bo vzpostavila serijska komunikacija.

Lastnosti `CtsFlowControl`, `DsrFlowControl`, `DtrFlowControl`, `DsrSensitivity`, `RtsControl` so namenjene hardverskemu nadzoru pretoka podatkov (ang. *Hardware flow control*), ki se pretežno uporablja v modemskih, printerskih in terminalskih povezavah. Programskega nadzora pretoka (ang. *Software flow control* ali *Xon / Xoff flow control*) objekt ne podpira, čeprav imajo Windows vgrajeno tudi to možnost.

12 Dolžina končnega bita je pravzaprav minimalni čas trajanja logičnega visokega nivoja na komunikacijskemu vodu, da sprejemnik to zazna kot konec podatkovnega okvira.

3.2.2 Objekt strežniškega mrežnega gonilnika



Slika 8: Hierarhija razreda TTCPServerDeviceDriver

`TTCPServerDeviceDriver` je razred, ki služi kot podlaga za izpeljavo različnih implementacij gonilnikov, kateri se obnašajo kot TCP strežniki. Izpeljan je iz razreda `TGenericDeviceDriver` (slika 8), implementira nekatere njegove metode in predstavi nove, ki poenostavijo delo s TCP vtičnicami (ang. *TCP sockets*).

Za dostop do Win32 vtičnic je uporabljena vmesniška knjižnjica (ang. *wrapper*) *Internet Component Suite* ali krajše *ICS* avtorja Francoisa Pietteja [15], ki omogoča objektni pristop k uporabi TCP vtičnic. ICS je prosto dostopen in ima popolnoma odprto kodo. Dovoljeno ga je uporabljati v komercialne namene in spreminjati njegovo delovanje, dokler so spremembe ustrezno označene. Omogoča uporabo protokolov TCP, UDP, SMTP, POP3, NNTP, HTTP, PING, TELNET, FINGER in mnogih drugih. Uporablja dogodkovno orientiran model programiranja (ang. *Event-driven*), vse metode pa delujejo kot neblokadne (ang. *Non-bloking*), kar pomeni, da se iz klicane metode vrnemo takoj po izvedbi in ne čakamo na odgovor nasprotne strani. V naboru knjižnic najdemo knjižnico `WSocketS`, ki predstavi objekt TCP strežnika `TWSocketServer`. Definicija razreda `TTCPServerDeviceDriver` je razvidna z izseka 3.

```
TSrvClient = class(TWSocketClient)
public
  fClientName: string;
  fBuffer: string;
end;

TTCPServerDeviceDriver = class(TGenericDeviceDriver)
private
  fSocketSrv: TWSocketServer;
  fPort: integer;
protected
  procedure ClientConnect(Sender: TObject; Client: TWSocketClient;
```

```
      Error: Word);
    procedure ClientDisconnect(Sender: TObject;
      Client: TWSocketClient; Error: Word);
    procedure ClientDataAvail(Sender: TObject; Error: Word);

    procedure StringAvail(Str: string; Client: TSrvClient);
      virtual; abstract;

    procedure Cycle; override;
  public
    constructor Create; reintroduce;
    destructor Destroy; override;

    procedure Initialize(_Link: TRemDev); override;
    procedure Finalize; override;

    function Stop(const timeout: Integer = 5000): Boolean; override;
  end;
```

Izsek 3: Definicija razreda TTCPServerDeviceDriver s pomožnim razredom TSrvClient

V konstruktorju `Create` se inicializira interni objekt TCP strežnika `fSocketSrv`. Ta objekt predstavlja vmesnik med mrežo in gonilnikom. Omogoča priklop več gostov, skrbi za pravilno inicializacijo TCP vtičnic, vzdrževanje povezave, pošiljanje in sprejemanje podatkov ter upravljanje z gosti. Vsak od gostov je predstavljen z instanco razreda `TSrvClient`, ki je izpeljana iz starševskega razreda `TWSocketClient`. Preko te instance lahko posameznemu gostu pošiljamo podatke in jih od njega sprejemamo. Omogoča pa še dostop do različnih podatkov o povezavi, kot na primer stanje povezave ali IP naslov gosta.

Destruktor `Destroy` privzame obnašanje predhodnika, poleg tega pa poskrbi za pravilno sprostitev objekta `fSocketSrv`.

Metoda `Initialize` privzame obnašanje svoje predhodnice ter doda obdelavo inicializacijskega niza, od koder razbere splošne nastavitve TCP strežnika. V trenutni implementaciji je to parameter, ki določa TCP vrata, na katerih naj strežnik posluša oziroma vzpostavlja povezave.

Metoda `Stop` poleg obnašanja predhodnice poskrbi še za pravilen odklop posameznega gosta in pravilno sprostitev strežniške vtičnice. V praksi se izkaže, da je za pravilno delovanje programa zelo pomemben pravilen izklop strežniške vtičnice, sicer operacijski sistem tej vtičnici določi stanje (TIME-WAIT), ki ga lahko ponastavimo samo z izklopom programa.

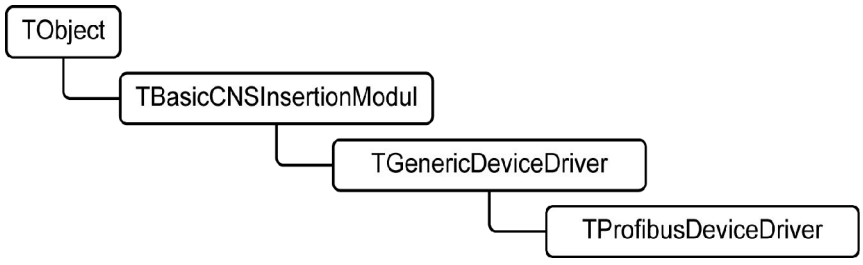
`Cycle` metoda je v tem primeru uporabljena za vključitev delovanja strežniške vtičnice in preverjanje delovanja le-te. V kolikor vtičnica ni več v stanju poslušanja, jo metoda ponovno poskuša postaviti v to stanje.

Metoda `ClientConnect` služi kot reakcija na dogodek in se kliče takrat, ko pride do povezave novega gosta na strežniško vtičnico. V njej se inicializira interne spremenljivke objekta gosta in določi metode za dogodke. Podobno se metodo `ClientDisconnect` sproži ob odklopu gosta.

`ClientDataAvail` je metoda, preko katere se sporoči, da je eden izmed priključenih gostov poslal podatke. V njej se prejete podatke prepiše v interno spremenljivko, se jih ustrezno obdela in poda metodi `StringAvail`, v kolikor je v njih dobljen razmejitveni niz. Ker je večina aktualnih komunikacij tekstovnega tipa z enim razmejitvenim znakom (navadno 13_{10} oz. $0D_{16}$), je tak način obdelave zelo primeren.

Metoda `StringAvail` je abstraktna metoda, ki jo mora vsaka funkcionalna oziroma dokončna izpeljava razreda `TTCPServerDeviceDriver` obvezno implementirati. Kliče se takrat, ko je na voljo niz znakov, zaključen z določenim razmejitvenim nizom.

3.2.3 Objekt profibus gonilnika



Slika 9: Hierarhija razreda TProfibusDeviceDriver

`TProfibusDeviceDriver` je izpeljava razreda `TGenericDeviceDriver` (slika 9). Za dostop do profibus vmesnika je uporabljen razred `TCP5613ProfiBusInterface`, ki služi kot vmesnik za dostop do sistemskih knjižnic Siemens CP5613 in CP5614 profibus vmesnikov. Komunikacija med profibus vozlišči (ang. *Nodes*) oziroma postajami (ang. *Stations*) je izvedena preko

DP protokola. Značilnost komunikacije, izvedene z DP protokolom, je ta, da ena ali več glavnih postaj (ang. *Master stations*) ciklično zajemajo podatke s priključenih podrejenih postaj (ang. *Slave stations*) ter jih ciklično vpisujejo in tako skrbijo, da imajo vsi udeleženci vedno pravo sliko podatkov.

Razred `TProfibusDeviceDriver` implementira nekatere metode svojega starša in doda novo lastnost `Profibus`, ki predstavlja programski vmesnik za dostop do profibus omrežja (glej izsek 4).

```
TProfibusDeviceDriver = class(TGenericDeviceDriver)
private
  fProfiBus: TCP5613ProfiBusInterface;
  fAccessPoint: string;
  procedure ProfiBusInfo(const Info: string);
public
  constructor Create; reintroduce;
  destructor Destroy; override;

  procedure Initialize(_Link: TRemDev); override;
  procedure Finalize; override;

  property ProfiBus: TCP5613ProfiBusInterface read fProfiBus;
end;
```

Izsek 4: Definicija razreda TProfibusDeviceDriver

Konstruktor `Create` poskrbi za pripravo profibus vmesnika `fProfiBus`, do katerega lahko nato dostopamo preko lastnosti `ProfiBus`. Vse izpeljanke tega razreda za komunikacijo uporabljajo to lastnost.

V destruktorju `Destroy` se poleg podedovanega obnašanja poskrbi še za pravilno sprostitev profibus vmesnika.

Metoda `Initialize` iz podanega inicializacijskega niza izlušči profibus dostopno točko (ang. *Access point*) in jo dodeli profibus vmesniku.

Struktura profibus vmesnika je razvidna iz izseka 5.

```
{ Siemens CP5613 & CP5614 ProfiBus interface }
TCP5613ProfiBusInterface = class(TObject)
private
  FSlaveAddress: integer;
  FAccessPoint: PChar;
  dpHandle: integer;
  dpData: ^DPR_CP5613_DP_S;
  dpData_ReadCount: integer;
  dpData_WriteCount: integer;
  FOnInfo: TInfoProc;
  FMaximumWaitTime: integer;
  FConnected: boolean;
```

```
    procedure SetAccessPoint(const Value: PChar);
    procedure SetSlaveAddress(const Value: integer);
    function ConvertError(dpErr: DP_ERROR): TProfiBusError;
    procedure SetOnInfo(const Value: TInfoProc);
    procedure Info(s: string);
    procedure SetMaximumWaitTime(const Value: integer);
    function WaitForState(state: integer; timeOut: integer):
      boolean;
    function GetSlaveAlive: boolean;
  public
    constructor Create; reintroduce;
    destructor Destroy; override;

    { procedure za delo s profibusom }
    function Connect(var err: TProfiBusError): boolean;
    function Disconnect(var err: TProfiBusError): boolean;

    function ResetDevice(var err: TProfiBusError): boolean;

    function ReadData(var Data: TDataArray;
      var err: TProfiBusError): boolean;
    function WriteData(const Data: TdataArray;
      var err: TProfiBusError): Boolean;

    { --- parametri ProfiBus-a --- }
    property AccessPoint: PChar read FAccessPoint
      write SetAccessPoint;
    property SlaveAddress: integer read FSlaveAddress
      write SetSlaveAddress;
    property MaximumWaitTime: integer read FmaximumWaitTime
      write SetMaximumWaitTime;
    property Connected: boolean read FConnected;

    property SlaveAlive: boolean read GetSlaveAlive;

    property OnInfo: TInfoProc read FOnInfo write SetOnInfo;
  end;
```

Izsek 5: Definicija razreda TCP5613ProfiBusInterface

Razred za svoje delo uporablja knjižnico DP_5613.DLL. Za naslavljanje te knjižnice je uporabljena C “header” datoteka DP_5613.H, prevedena v Object Pascal jezik.

Ob klicu konstruktorja `Create` se izvede nalaganje knjižnice. V kolikor nalaganje ni uspešno, ker knjižnica ne obstaja, oziroma je napačno nameščena ali je napačna verzija, instanca razreda ne more delovati in sproži napako.

Destruktor razreda `Destroy` izvede odklop vmesnika in sprosti dostop do uporabljenih DLL knjižnic.

`Connect` metoda izvede inicializacijo CP561x vmesnika po specificiranem zaporedju, ki ga podaja proizvajalec in zažene komunikacijo. V primeru, da tega ne more storiti, je rezultat metode `False` in parameter `err` nosi informacijo o napaki,

zaradi katere ni bilo mogoče izvesti zagona. Inicializacija je izvedena po naslednjem zaporedju:

1. zagon profibus kartice: `DP_start_cp(FAccessPoint, nil, dpErr);`
2. pridobitev kontrolne ročice: `DP_open(FAccessPoint, dpHandle, dpErr);`
3. pridobitev kazalca na podatkovno strukturo vmesnika: `DP_get_pointer (dpHandle, FMaximumWaitTime, p, dpErr);`
4. postavitev komunikacije v stanje "ustavljeno": `DP_set_mode (dpHandle, DP_STOP, dpErr);`
5. praznjenje izhodnega pomnilnika: `DP_set_mode(dpHandle, DP_CLEAR, dpErr);`
6. postavitev vmesnika v operativno stanje: `DP_set_mode(dpHandle, DP_OPERATE, dpErr).`

Po uspešno opravljenem klicu metode je potrebno počakati še nekoliko časa, preden se začne brati ali pisati podatke (čas je odvisen od števila priključenih vozlišč in hitrosti komunikacijskega voda), tako da glavna postaja dobi realno sliko trenutnega stanja postaj v omrežju in pridobi vrednosti njihovih izmenjevalnih registrov.

`Disconnect` metoda opravi zaključno sekvenco prekinitve komunikacije. Sekvenca je realizirana na podlagi specifikacije proizvajalca in ima naslednje stopnje:

1. praznjenje izhodnega pomnilnika: `DP_set_mode(dpHandle, DP_CLEAR, dpErr);`
2. ustavitev komunikacije: `DP_set_mode(dpHandle, DP_STOP, dpErr);`
3. prehod v odključeno stanje: `DP_set_mode(dpHandle, DP_OFFLINE, dpErr);`
4. sprostitev kazalca na podatkovno strukturo: `DP_release_pointer (dpHandle, dpErr);`
5. zapiranje ročice vmesnika: `DP_close(dpHandle, dpErr);`
6. resetiranje profibus kartice: `DP_reset_cp(PChar(FAccessPoint), dpErr).`

Po klicu te metode ni več možno brati ali pisati podatkov.

Preko metode `ResetDevice` lahko neposredno izvedemo resetiranje profibus kartice. Metoda vrača `True`, če je resetiranje uspešno, v nasprotnem primeru je rezultat `False`. Dodatne informacije o napaki dobimo preko parametra `err`.

Metoda `ReadData` je namenjena branju podatkov, metoda `WriteData` pa pisanju podatkov v priključene postaje. Na katero postajo se branje oz. pisanje nanaša, določimo preko lastnosti `SlaveAddress`.

Lastnost `AccessPoint` določa profibus dostopno točko in jo moramo določiti pred klicem metode `Connect`. Dostopna točka je ime konfiguracijske datoteke, ki je shranjena v profibus kartici in pove nastavitve mreže, nastavitve priključenih postaj in količino prenašanih podatkov glede na vsako izmed postaj.

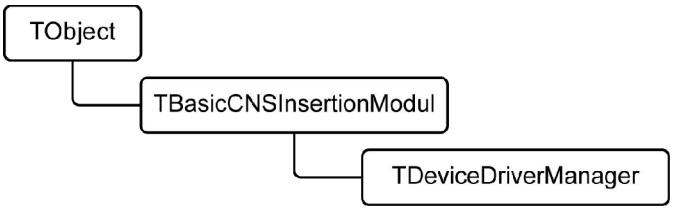
Lastnost `MaximumWaitTime` določa, koliko časa naj se čaka na rezultat klicanih metod iz DP_5613 knjižnice v milisekundah. V trenutni implementaciji to uporablja samo metoda `DP_get_pointer`.

Lastnost `Connected` bo nastavljena na `True`, če je bila inicializacijska sekvenca izvedena uspešno (klic metode `Connect`), oziroma `False`, če inicializacijska sekvenca ni bila izvedena, je bila izvedena neuspešno ali je bila klicana metoda `Disconnect`.

Lastnost `SlaveAlive` nam pove, če je povezava na postajo (številka postaje je določena z lastnostjo `SlaveAddress`) vzpostavljena. Lastnost ima vrednost `True`, v kolikor je postaja v podatkovni strukturi profibus vmesnika vpisana s statusom `DPR_SLV_READY`, in vrednost `False` v vseh drugih primerih.

3.3 Razred upravljalnika gonilnikov

Objekt upravljalnika gonilnikov je v programu en sam. Ime razreda je `TDeviceDriverManager` in izhaja iz razreda `TBasicCNSInsertionModul` (slika 10).



Slika 10: Hierarhija razreda TDeviceDriverManager

V njem so predstavljene metode za upravljanje z gonilniki, poskrbi pa tudi za pravilno upravljanje dogodkov iz gonilnikov. Definicija razreda je razvidna z izseka 6.

```
  TDeviceDriverManager = class(TBasicCNSInsertionModul)
  private
    fDriverList: TList;
    fCommonEventList: TList;
    csEventListSync: TCriticalSection;
    fEventThread: TEventThread;
    fEnabled: boolean;
    fOnEvent: TDriverEventMessage;
    fOnStatusChange: TDriverStatusChange;

    procedure DriverEvent(Driver: TGenericDeviceDriver; Event:
THoldEvent);
    procedure DriverStatusChange(Driver: TGenericDeviceDriver);

    function GetCount: integer;
    function GetDriver(index: integer): TGenericDeviceDriver;
    procedure SetEnabled(const Value: boolean);
    procedure SetOnEvent(const Value: TDriverEventMessage);
    function GetEventCount: integer;
  public
    constructor Create; reintroduce;
    destructor Destroy; override;

    procedure Initialize(Devices: TRemDevSet);
    procedure Finalize;

    function Add(Link: TRemDev): TGenericDeviceDriver;
    procedure Clear;

    function Find(Link: TRemDev): TGenericDeviceDriver; overload;
    function Find(ID: integer): TGenericDeviceDriver; overload;

    { lock in unlock dolocata v kaksnem stanju je csEventListSync }
    procedure Lock;
    procedure Unlock;

    { ko se dobi nove podatke in je potrebno dolociti nove driverje,
      je potrebno poskrbeti, da se noben dogodek ne izgubi }
    procedure BeginReassign;
    procedure EndReassign;

    property Enabled: boolean read fEnabled write SetEnabled;
    property Driver[index: integer]: TGenericDeviceDriver
```

```
      read GetDriver; default;
    property Count: integer read GetCount;

    property EventCount: integer read GetEventCount;
    property OnEvent: TDriverEventMessage read FOnEvent write
SetOnEvent;
    property OnStatusChange: TDriverStatusChange read
fOnStatusChange
      write fOnStatusChange;
  end;
```

Izsek 6: Definicija razreda TDeviceDriverManager

Konstruktor `Create` inicializira interne spremenljivke in kritične odseke ter poskrbi za zagon programske niti za obdelavo dogodkov. Destruktor `Destroy` kliče metodo `Finalize`, nato pa sprosti pomnilnik, ki je bil zavzet pri klicu konstruktorja.

Metoda `Initialize` sprejme kot parameter seznam gonilnikov. Klic metode sprosti vse do tedaj prisotne gonilnike in naloži vse, ki so našteti v parametru `Devices` in niso navedeni kot izključeni (status gonilnika `rdsDisabled`). Nalaganje gonilnikov je izvedeno s klicem metode `Add`.

Metoda `Finalize` sprosti vse naložene gonilnike s klicem metode `Clear`, izključi nit za obdelavo dogodkov in sprosti vse morebitne neobdelane dogodke. Pri njeni uporabi je potrebno biti pazljiv, saj lahko pride do izgube podatkov iz naslova prenosa na višji nivo.

Metodo `Add` uporabimo takrat, ko želimo dodati nov gonilnik na podlagi strukture `TRemDev`. Metoda poskrbi, da se glede na tip naprave naloži pripadajoči gonilnik, določi parametre in kliče gonilnikovo metodo `Initialize`.

`Clear` metoda sprosti vse gonilnike, naložene v objekt prek klica metode `Add`. Po klicu te metode je ustavljena komunikacija na vse priključene naprave, morebitni neobdelani dogodki pa ostanejo v sistemu. Metoda je namenjena zamenjavi vseh podatkov o napravah.

Metodi `Find` sta namenjeni iskanju gonilnikov na podlagi podane podatkovne strukture ali na podlagi identifikacijske številke ID. Rezultat metod je objekt tipa `TGenericDeviceDriver`, torej objekt gonilnika, ki ustreza iskanemu kriteriju. Poudariti je potrebno, da oba iskalna pogoja enolično določata en sam gonilnik. Direktiva `overload` omogoča enako poimenovanje več različnih metod (metod, ki

se razlikujejo v številu in / ali tipu parametrov), vendar morajo vse vračati rezultat enakega tipa.

Metodi `Lock` in `Unlock` nadzirata prehode v kritični odsek `csEventListSync` in iz njega. Ta odsek je uporabljen pri sinhronizaciji obdelave dogodkov iz gonilnikov.

Metodi `BeginReassign` in `EndReassign` uporabljamo, ko pride do spremembe konfiguracije oddaljenega komunikatorja. `BeginReassign` dogodke razveže povezave na gonilnik, kjer je nastal, `EndReassign` pa to povezavo ponovno vzpostavi. Vsakemu klicu metode `BeginReassign` mora obvezno slediti klic metode `EndReassign`. Klici ne smejo biti kumulativni, kar pomeni, da ne smemo dvakrat klicati `BeginReassign` in nato dvakrat `EndReassign`. Pri uporabi teh dveh metod je potrebno zagotoviti, da se vmes ne izvaja obdelava dogodkov iz gonilnikov. V ta namen lahko uporabimo metodi `Lock` in `Unlock`.

Lastnost `Enabled` služi za hkratni zagon (dodelitev vrednosti `True`) ali zaustavitev vseh gonilnikov (dodelitev vrednosti `False`). Pri hkratnem zagonu se kličejo `Run` metode gonilnikov, ki so nastavljeni tako, da se jih ob nalaganju avtomatsko zažene. Ob zaustavitvi se kliče `Stop` metode vseh gonilnikov.

Privzeta lastnost `Driver` je namenjena indeksiranemu dostopu do naloženih gonilnikov. Začetni indeks je 0, končni pa je odvisen od števila gonilnikov. Število naloženih gonilnikov nam vrne lastnost `Count`.

Lastnost `EventCount` je informativnega značaja in vrne število neobdelanih dogodkov iz gonilnikov.

Dogodek `OnEvent` se proži takrat, ko je dogodek iz gonilnika obdelan in ga je potrebno poslati preko mreže na višji nivo. Dogodek `OnStatusChange` je vezan na spremembo statusa gonilnika.

3.4 Povezanost in delovanje programskih objektov

3.4.1 Zamisel večnitnih procesov

Sodoben operacijski sistem podpira zaporedno izvajanje več neodvisnih programskih blokov. Vsakemu takemu bloku zato priredimo eno nit (ang. *Thread*), preko katere nato operacijski sistem upravlja izvajanje. Od tod pojem večnitnosti (ang. *Multithreading*) [16].

V Windows operacijskih sistemih je potrebno najprej obravnavati dva pomembna pojma: proces in nit. Proces je sestavljen iz pomnilnika in virov. Pomnilnik je razdeljen v tri dele: sklad, podatki in programska koda.

V skladu so shranjene vse lokalne spremenljivke in klicne reference (ang. *Call stack*). Vsaki niti se dodeli njej lasten sklad.

Podatki so sestavljeni iz spremenljivk, ki niso lokalne, in pomnilnika, ki se ga zavzame dinamično.

Programska koda je izvršni del programa, namenjen samo branju.

Vsi trije deli so na voljo vsem nitim znotraj enega procesa. Z vidika operacijskega sistema lahko proces vsebuje samo pomnilnik in niti. Vsak proces ima svojo identifikacijsko številko, ki je v sistemu edinstvena.

Izvršilna nit se začne, ko procesor začne z izvajanjem dela programske kode. Nit je v lasti procesa, ta pa lahko vsebuje večje število niti. Število niti na proces je v praksi potrebno omejiti, na kar vpliva število procesorjev v računalniku, zahtevnost posameznih niti glede procesorskega časa, obremenjenost računalnika, itd. Večje število niti vodi v zmanjšanje odzivnosti procesorja, saj ima več opravka z upravljanjem niti kot pa z njihovim izvrševanjem. Vsaka izmed niti ima lasten sklad, ročice za upravljanje napak (ang. *Exception handlers*) in izvršilno prioriteto.

Prioriteta niti je sestavljena iz prioritetnega razreda procesa, v katerem teče, in lastnega prioritetnega nivoja. Skupaj sestavljata lestvico med 0 in 31, kjer 0 pomeni najnižjo prioriteto in 31 najvišjo. Najvišje prioritete je potrebno uporabljati zelo previdno, saj lahko blokirajo izvrševanje drugih niti. Najvišjo prioriteto, 31, navadno

ni priporočljivo uporabljati, ker blokira tudi sistemske niti, kot so niti za obdelavo vnosa prek miške in tipkovnice.

3.4.2 Problematika sinhronizacije mednitnih komunikacij

Če uporabljamo v programu več niti, često pridemo do problematike usklajevanja branja in pisanja skupnih podatkov. Windows operacijski sistem v ta namen nudi več sinhronizacijskih objektov:

- kritični odseki (ang. *Critical sections*): obnašajo se kot vrata, ki preprečujejo drugim nitim dostop do dela izvršne kode. Za razliko od drugih sinhronizacijskih objektov se kritične odseke sme uporabljati samo znotraj enega procesa. Dovoljujejo dostop samo eni niti hkrati do zaščitenih resursov ali pomnilnika, kar lahko ob napačni uporabi vodi do resne degradacije izvršne opravilnosti znotraj procesa;
- semaforji (ang. *Semaphores*): uporabljajo se v primerih, ko lahko do nekega resursa hkrati dostopa določeno (omejeno) število niti. Objekt semaforja lahko uporabljamo za sinhronizacijo niti tudi med procesi;
- mutex (iz ang. *MUTual EXclusion* – vzajemno izključevanje): sinhronizacijski objekt, ki dovoljuje dostop do zaščitenega resursa samo eni niti hkrati. Uporabljamo ga lahko za sinhronizacijo niti med procesi in znotraj procesa. Načeloma opravlja enako funkcijo kot kritični odsek, vendar je procesorsko bolj zahteven, predvsem na račun medprocesne združljivosti;
- dogodki (ang. *Events*): sinhronizacijski dogodek je način sinhronizacije, ki se uporablja pri nitih, katerih izvrševanje je medsebojno pogojeno.

V programskem okolju Delphi je za mednitno sinhronizacijo najenostavneje uporabiti instanco razreda `TCriticalSection` (unit `SyncObjs.pas` – glej izsek 7). Ta sicer uporablja API funkcije kritičnih odsekov (`InitializeCriticalSection`, `DeleteCriticalSection`, `EnterCriticalSection`, `LeaveCriticalSection`), vendar je njihova uporaba predstavljena kot upravljanje z objektom.

```
TCriticalSection = class(TSynchroObject)
protected
  FSection: TRTLCriticalSection;
```

```
public
  constructor Create;
  destructor Destroy; override;
  procedure Acquire; override;
  procedure Release; override;
  procedure Enter;
  procedure Leave;
end;
```

Izsek 7: Definicija razreda TCriticalSection

Sinhronizacija je izvedena s klicanjem metod `Acquire` in `Release` oziroma `Enter` in `Leave`. Pred delom programske kode, do katerega želimo sinhronizirati dostop (oz. preprečiti hkratni dostop), kličemo metodo `Acquire` ali `Enter` (uporabljena je API funkcija `EnterCriticalSection`). V kolikor je izvajanje neke niti že znotraj kritičnega odseka, bo izvajanje druge niti blokirano na mestu klica ene izmed teh metod, vse dokler prva ne bo klicala metode `Release` ali `Leave` (uporabljena API funkcija `LeaveCriticalSection`). Paziti je potrebno, da se po vsakem klicu metod vstopa v kritični odsek kliče tudi metode izstopa, v nasprotnem primeru lahko pride do t.i. smrtnega objema (ang. *Deadlock*), ko ena nit večno blokira izvajanje drugih niti. Precej pogost je še drug primer smrtnega objema, ko je izvajanje dveh niti zaustavljeno zaradi čakanja druge na drugo, ki je prav tako posledica napačne uporabe sinhronizacijskih objektov. Za preprečevanje takih situacij Microsoft priporoča uporaba virtualnih prioritet dostopa do zaščitenih resursov. To pomeni, da moramo za dostop do resursa vstopiti v toliko kritičnih odsekov, kolikor jih je prioritetno nad njim, pri čemer vsakemu resursu pripada lasten kritični odsek.

Pomembna lastnost kritičnih odsekov je tudi ta, da lahko ena nit večkrat kliče vstop v kritični odsek(možnost kumulativnega klicanja), pri tem pa njeno izvajanje ni blokirano. Ta lastnost je zelo dobrodošla pri uporabi rekurzivnih funkcij, ko lahko funkcija ponovno kliče samo sebe. Vendar je v tem primeru potrebno klicati izstop iz kritičnega odseka tolikokrat kot vstop, saj se šele po zadnjem izstopu vstop dovoli tudi drugim nitim.

3.4.3 Delovanje gonilnika

Izvajanje gonilnika poteka v njemu lastni niti. Dejansko to pomeni, da so klici metode `Cycle` izvedeni v posebnem objektu tipa `TCycleThread` (glej izsek 8).

```
TCycleThread = class(TThread)
protected
  fThreadRunning: boolean;
  fOwner: TGenericDeviceDriver;
  procedure Execute; override;
public
  constructor Create(_Owner: TGenericDeviceDriver); reintroduce;
  destructor Destroy; override;
end;
```

Izsek 8: Definicija razreda TCycleThread

Razred `TCycleThread` je izpeljan iz razreda `TThread`. Ta je del programskega okolja Delphi in omogoča relativno enostavno implementacijo programske niti. Njegov konstruktor `Create` je dopolnjen in preko parametra `_Owner` sprejme kazalec na objekt gonilnika, za katerega izvajanje naj skrbi.

Implementacija metode `Execute` je obvezna, saj je prav ta nosilec niti in je v izvornem razredu definirana abstraktno. Izvajanje metode je dejansko izvajanje niti, za kar skrbi razred `TThread`. Periodično klicanje metode `Cycle` je izvedeno znotraj metode `Execute`, s čimer dosežemo, da se izvajanje metode `Cycle` vrši v lastni niti, neodvisni od izvajanja drugih gonilnikov.

Drugi programski konstrukt, ki ga je na tem mestu potrebno omeniti, je razred `TEventThread`, ki skrbi za obdelavo dogodkov iz gonilnika (glej izsek 9). Izpeljan je iz razreda `TThread`, njegova naloga pa je razbremenitev komunikacijske niti posameznega gonilnika tako, da nase prevzame obdelavo dogodkov in razpošiljanje dogodkov iz naprav preko mreže na centralno zbirno mesto.

```
TEventThread = class(TThread)
private
  fOwner: TDeviceDriverManager;
  fCommonSync: TCriticalSection;
  fOnEvent: TDriverEventMessage;
  fCommonList: TList;
  fFinished: boolean;
protected
  procedure Execute; override;
public
  constructor Create(_Owner: TDeviceDriverManager); reintroduce;
  destructor Destroy; override;

  property Finished: boolean read fFinished;
```

```
    property CommonList: TList read fCommonList write fCommonList;
    property CommonSync: TCriticalSection read fCommonSync write
fCommonSync;
    property OnEvent: TDriverEventMessage read fOnEvent write
fOnEvent;
  end;
```

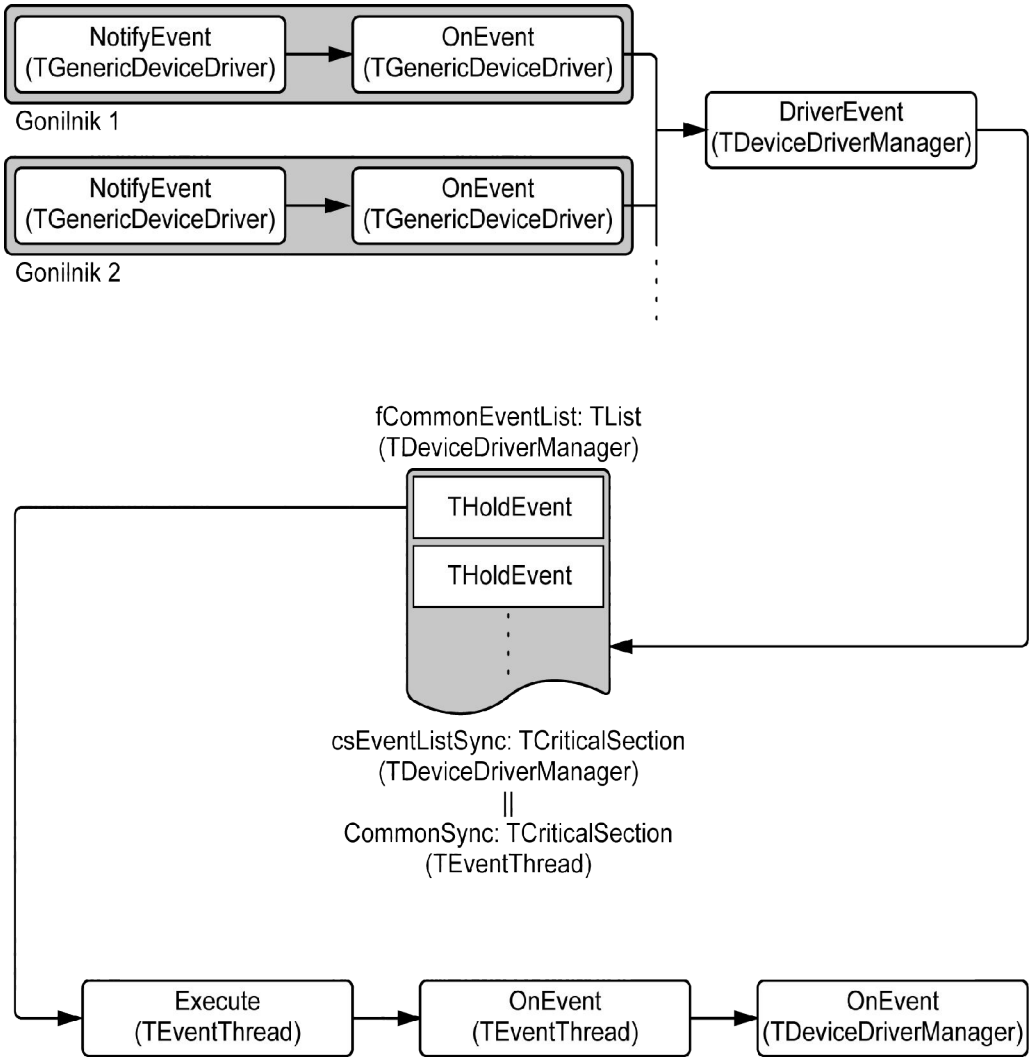
Izsek 9: Definicija razreda TEventThread

Lastnost `CommonList` je kazalec na instanco razreda `TList`. Predstavlja skupen seznam dogodkov na oddaljenem komunikatorju. Skupni seznam je uporabljen zaradi boljšega redosledja pojavov posameznega dogodka glede na gonilnik in poenostavitve obdelave le-teh. Do seznama dostopa hkrati mnogo niti: komunikacijske niti gonilnikov, nit obdelave dogodkov in glavna nit programa. Glede na to je nujno potrebno uvesti sinhronizacijo dostopa do seznama. Za njo je uporabljena metoda kritičnih odsekov. Ker mora biti referenca na objekt kritičnega odseka skupna za vse niti, objekt kreiramo v glavni niti in nato propagiramo na preostale niti. Tako razred `TEventThread` sprejme kazalec na objekt kritičnega odseka preko lastnosti `CommonSync`.

Lastnost `Finished` nosi vrednost `True`, če je izvajanje niti obdelave končano, in vrednost `False`, če nit še ni bila zagnana ali pa je njeno izvajanje v teku.

Programski dogodek `OnEvent` se proži takrat, ko je dogodek iz gonilnika obdelan in ga je potrebno poslati preko mreže. Podrobnejši proceduralni potek je razviden na sliki 11.

Slika 11 prikazuje potek prenosa dogodka iz gonilnika od nastanka do zahteve po razpošiljanju. Glede na komunikacijski protokol gonilnik ugotovi, da je na priključeni napravi prišlo do novega dogodka. Ko pridobi zadostno količino informacij, kliče metodo `NotifyEvent`, ki je implementirana v izvornem razredu `TGenericDeviceDriver`. Da lahko opravi klic te metode, mora pridobiti vsaj naslov elementa, na katerem je prišlo do nekega dogodka, in tip dogodka oziroma naslov in novo vrednost neke opazovane veličine.



Slika 11: Princip prenosa dogodka iz gonilnika

Metoda `NotifyEvent` opravi klic programskega dogodka `OnEvent`. Njemu je prirejena metoda `DriverEvent`, ki se vsakemu gonilniku priredi ob kreiranju instance programskega objekta, torej pri klicu metode `Add` instance razreda `TDeviceDriverManager`, in je skupna vsem gonilnikom. Metoda `DriverEvent` poskrbi za sinhronizirano vpisovanje v skupni seznam dogodkov `fCommonEventList` preko kritičnega odseka `csEventListSync`. Kritični odsek nato uporablja `Execute` metoda instance razreda `TEventThread` za branje in odstranjevanje dogodkov s seznama.

Na skupni seznam se dogodki vpisujejo v začasnem, podatkovno neodvisnem zapisu,

ki ga predstavljajo objekti razreda `THoldEvent`. Šele `Execute` metoda razreda `TEventThread` preveri, ali je element, na katerega se dogodek nanaša, prisoten v sistemu, ali naj se na njem javlja dogodke podanega tipa in ali je podana vrednost opazovane veličine. V kolikor se zadosti pogojem, se dogodek iz začasne strukture prevede v podatkovno odvisno strukturo `TEventMessage`, preko katere je omogočen hitrejši in strukturiran dostop do podatkov relevantnih za dani dogodek. V nasprotnem primeru, torej ko se pogojem ne zadosti, se dogodek zavrže in sprosti njegovo mesto v pomnilniku.

Ko je dogodek obdelan in pripravljen za razpošiljanje, se sproži programski dogodek `OnEvent`, ki ga instanca razreda `TDeviceDriverManager` veže neposredno na svoj programski dogodek `OnEvent`. Na tega je nato vezan koordinator, ki skrbi za posredovanje na objekt mrežnega upravljanja in od tam na centralno zbirno mesto, to je v program DTS.

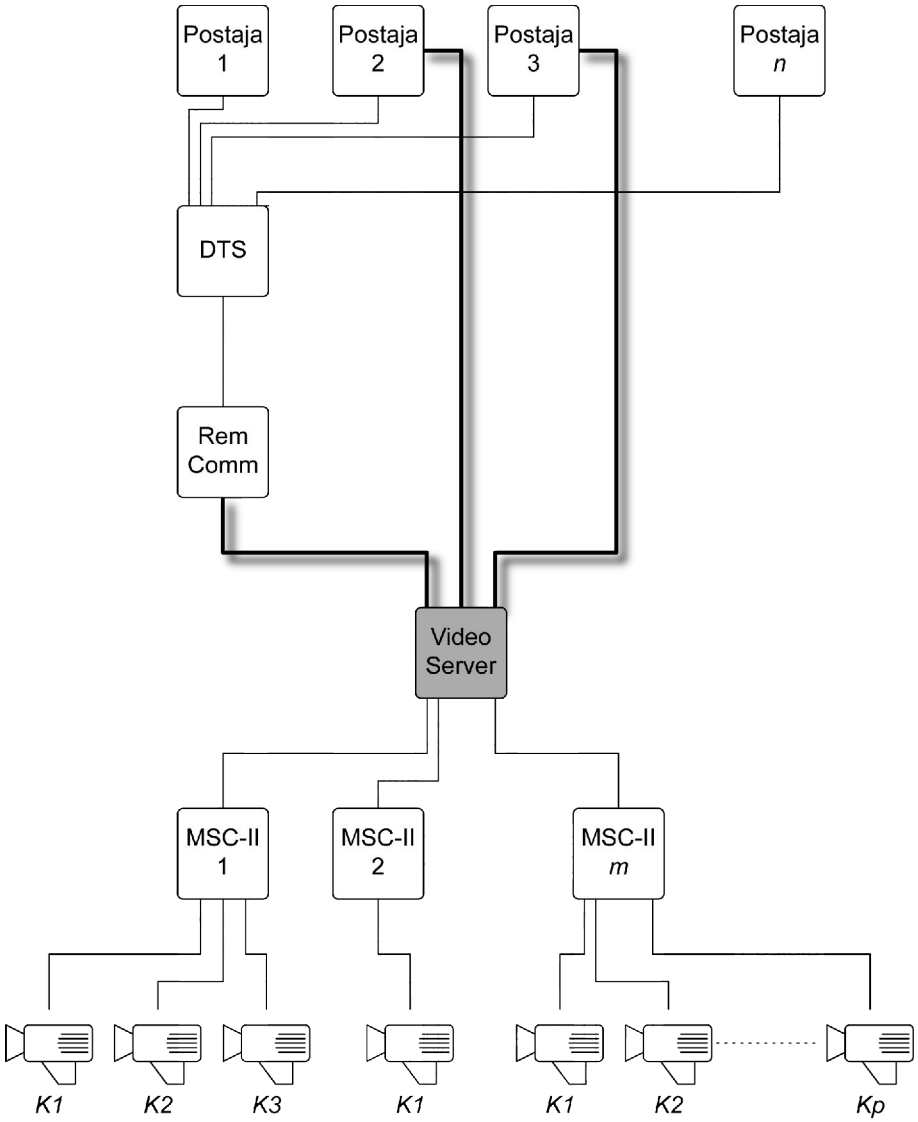
4. Primeri realizacije gonilnikov

V tem poglavju je predstavljena konkretna implementacija gonilnikov na opisanem programskem okvirju. Od tridesetih dosedaj realiziranih gonilnikov so bili izbrani trije, ki bolj ali manj karakterizirajo možnosti opisanega programskega okvirja.

4.1 Video Server – Geutebrueck Multiscope

Geutebrueck je nemško podjetje, ki se primarno ukvarja s sistemi za video nadzor . Njihov vodilni izdelek je digitalni sistem za prikaz, shranjevanje in upravljanje video posnetkov, imenovan Multiscope II [17]. Grobo gledano, Multiscope sistem sestavlja SQL strežnik, na katerega so priključene video nadzorne kamere preko video vmesnika in uporabniški program, ki omogoča upravljanje s sistemom.

Ker so lahko video nadzorni sistemi sestavljeni iz večjega števila kamer, je potrebno biti pozoren tudi na procesne zmožnosti strežnika. Navadno imajo večji objekti več video strežnikov, ki se tudi navzven obnašajo samostojno. Ker je s funkcionalnega vidika cel sistem smiselno obravnavati kot eno enoto, je bil v ta namen razvit program Integra VideoServer. Ta je pravzaprav neke vrste koncentrator, saj iz ene točke omogoča dostop do poljubnega Multiscope strežnika, če mu je le ta poznan in dovoli povezavo.



Slika 12: Povezovanje na video sistem Multiscope preko programa VideoServer

Zamišljeno delovanje VideoServer koncentratorja je razvidno s slike 12. Odebeljeno so označene povezave programov paketa CNS na storitve VideoServerja. Oddaljeni komunikator (na sliki označen kot RemComm) uporablja VideoServer za zajem podatkov, ki nosijo informacije o zaznavi premika (ang. *Activity detection*), izpadu sinhronizacijskega signala na kameri in izpadu video signala.

Nadzorne postaje (na sliki označene kot Postaja *x*) uporabljajo VideoServer za dostop do trenutnih slik, zajetih iz video kamer, ter za dostop do arhivskih posnetkov, pri katerih je gonilnik javil nek dogodek, npr. zaznal premik na sliki v coni 2.

VideoServer deluje hkrati kot strežnik za programe paketa CNS ter kot gost za strežnike Multiscope. Celotna komunikacija je izvedena preko TCP/IP protokola. Gonilnik mora torej biti pisan kot TCP gost. Kot je razvidno iz opisa vloge VideoServer-ja, je kompleksnost TCP/IP komunikacije zaradi zahtevanih hitrih, vzporednih prenosov slik velika. Če temu prištejemo še dejstvo, da moramo ta komunikacijski protokol uporabiti še v programu nadzorne postaje, je smiselno napisati programski razred, ki skrbi izključno za komunikacijo na VideoServer. Ta razred je bil tudi uporabljen pri implementaciji gonilnika za VideoServer.

```
  TVSElement = class(TObject)
  private
    fServer: boolean;
    fAD4: integer;
    fSync: integer;
    fVideoOut: integer;
    fAD2: integer;
    fAD3: integer;
    fAD1: integer;
    fMSCName: string;
    fLink: TDevElement;
    fStatus: TStatusi;
    procedure SetLink(const Value: TDevElement);
  public
    property Link: TDevElement read fLink write SetLink;
    property MSCName: string read fMSCName write fMSCName;
    property Server: boolean read fServer write fServer;
    property Status: TStatusi read fStatus write fStatus;
    property AD1: integer read fAD1 write fAD1;
    property AD2: integer read fAD2 write fAD2;
    property AD3: integer read fAD3 write fAD3;
    property AD4: integer read fAD4 write fAD4;
    property Sync: integer read fSync write fSync;
    property VideoOut: integer read fVideoOut write fVideoOut;
  end;

  TDevice_VideoServer = class(TGenericDeviceDriver)
  private
    fHost: string;
    fPort: integer;
    fClient: TTCPVideoClient;
    fInternalList: TList;
    { procedure za spremljanje VS-ja }
    procedure VSEventAvailable(Sender: TObject; const EventStart:
boolean;
      const EventType, EventTypeID: integer; const MSCName: string);
    procedure VSConnectedChange(Sender: TObject; const connected:
```

```
boolean);
    procedure VSMSCListAvail(Sender: TObject; const Napaka: boolean;
      Lista: TList);
    procedure VSMSCStatChange(Sender: TObject; const MSCName:
string;
      const Status: TStatusi);
    procedure VSError(sender: TObject);

    function Add(_Link: TDevElement): TVSElement;
    function FindServer(const SrvName: string): TVSElement;
    procedure Clear;
  public
    constructor Create; reintroduce;
    destructor Destroy; override;

    procedure Cycle; override;

    function GetDriverVersion: TModulVersion; override;
    function GetDriverInfo: string; override;

    procedure Initialize(_Link: TRemDev); override;
    procedure Finalize; override;

    function Stop(const timeout: Integer = 5000): Boolean; override;

    procedure ExecuteCommand(const Command: String); override;
  end;
```

Izsek 10: Definicija razreda TDevice_VideoServer in pomožnega razreda TVSElement

Razred gonilnika se imenuje `TDevice_VideoServer` in je izpeljan iz razreda `TGenericDeviceDriver` (glej izsek 10). Konstruktor razreda `Create` je dopolnjen tako, da poskrbi za inicializacijo interne spremenljivke `fClient` tipa `TTCPVideoClient`, ki skrbi za komunikacijo na `VideoServer`, ter internega seznama elementov `fInternalList`. Njegovi vnosi so zapisani v instancah razreda `TVSClient`. Pomembno je, da se spremenljivko `fClient` inicializira prav v tem delu, ki je klican iz glavne niti, saj je njeno delovanje vezano na t.i. sporočilno vrsto (ang. *Message queue*). Ker Delphi avtomatsko implementira obdelovanje sporočilne vrste za glavno nit, v Windows okolju pa mora imeti vsaka nit svojo obdelavo sporočilne vrste, je pripravno uporabiti glavno nit za usmerjanje sporočil TCP/IP komunikacije. Na ta način se izognemo pisanju dodatne kode, ki bi skrbela za obdelavo sporočilne vrste znotraj posamezne niti, ne da bi pri tem bistveno obremenili glavno nit.

Razred `TVSClient` je optimizirana predstavitev posameznih naslovov elementov, ki so v sistemu določeni kot elementi VideoServer gonilnika. Optimizacija je

izvedena na račun za računalniško obdelavo neprimernega zapisa naslovov elementov, ki imajo naslednjo obliko:

<ime Multiscope strežnika>,<ID cone 1>, <ID cone 2>,<ID cone 3>,<ID cone 4>,<ID sinhronizacije>,<ID video signala>

Ta polja se ob določitvi `Link` lastnosti razreda `TVSClient` prepišejo iz naslovnega niza v ločene spremenljivke, kar omogoča hitrejše iskanje. Komunikacijski protokol namreč sporoča samo posamezne unikatne ID številke, na katerih je prišlo do začetka ali konca dogodka. Glede na ID številko in njeno pozicijo v naslovnem nizu lahko sklepamo, ali je bil alarm, ki ga je sprožila detekcija premikov (polja con od 1 do 4 v naslovnem nizu), javljanje izpada sinhronizacije (polje sinhronizacije) ali javljanje izpada video signala (polje video signala).

Destruktor `Destroy` razreda `TDevice_VideoServer` poskrbi za zaustavitev komunikacije in sprostitve vsega zajetega pomnilnika.

Metoda `Cycle` je implementirana tako, da skrbi za zagon komunikacije oziroma za ponovno vzpostavitev morebiti padle povezave. Ob vsakem ciklu pogleda, ali objekt `fClient` povezavo vodi kot aktivno ali neaktivno. Če je povezava neaktivna (to velja tudi pri prvem zagonu), sproži poskus povezave. Ker se kontrola vrši periodično, lahko hitro zaznamo izpade in v čim krajšem času povezavo ponovno vzpostavimo.

Metodi `GetDriverInfo` in `GetDriverVersion`, kot je postavljena zahteva v izvornem razredu, vračata opis gonilnika (*Integra VideoServer driver ©2004 Robotina d.o.o.*) in verzijo implementacije gonilnika (v3.0).

Metoda `Initialize` dopolni podedovano obnašanje z branjem inicializacijskega niza, od koder izlušči nastavitve, potrebne za povezavo na VideoServer. Znotraj te metode je izvedena tudi optimizacija oziroma inicializacija optimiziranega zapisa podatkovne strukture elementov.

Metoda `Finalize`, poleg podedovanega obnašanja, poskrbi še za sprostitev vnosov optimiziranega zapisa podatkov. Po klicu te metode je gonilnik v stanju, kot je bil tik pred inicializacijo.

Metoda `Stop` je predstavljena že v izvornem razredu, vendar je tu njeno delovanje dopolnjeno tako, da poskrbi tudi za izključitev komunikacije na VideoServer. Pomembno je, da pred prekinitvijo komunikacije zaključimo z izvajanjem komunikacijske niti gonilnika, ki skrbi za ponovno vzpostavitev komunikacije v primeru prekinitev. Če tega ne storimo, nimamo nobenega zagotovila, da bo po klicu metode `Stop` komunikacija tudi dejansko ustavljena.

Ker je možno na VideoServer prožiti tudi določene zahteve, je tu implementirana, sicer neobvezna metoda `ExecuteCommand`. Ta kot parameter sprejme niz znakov v obliki:

<Ime Multiscope strežnika>|<ID tipa>|<Čas snemanja>|<Število slik>|<Tip dogodka>|

Pomen posameznih polj je naslednji:

- ime Multiscope strežnika: na kateri strežnik se zahteva nanaša;
- ID tipa: interna nomenklatura pri Multiscope produktih (ang. *Type ID*), ki lahko označuje posamezno kamero ali neko vnaprej določeno zaporedje kamer. Pomen je odvisen od nastavitev Multiscope strežnika;
- čas snemanja: koliko časa, podano v milisekundah, naj se vrši snemanje. Če je podana vrednost 0, se uporabi preddoločeno vrednost, ki je nastavljena za določen ID tipa. To polje se izključuje z naslednjim, tako da je lahko od nič različno največ eno izmed njiju;
- število slik: koliko slik naj se posname. Ker je na Multiscope strežniku vnaprej določeno, v kakšnih časovnih intervalih naj zajema slike, je tudi čas snemanja ob podaji števila slik enolično določen. Zaradi tega ne smemo hkrati podati zahtevanega časa snemanja in zahtevanega števila slik, saj se ta dva podatka medsebojno izključujeta;
- tip dogodka: dodatna informacija, zakaj je bil dogodek prožen. Podatek je zgolj informativne narave.

Ko se ukaz posreduje VideoServerju, ta proži začetek snemanja na določenem Multiscopu. Konec snemanja je določen bodisi z enim od parametrov časa snemanja ali števila slik bodisi z preddoločenimi vrednostmi na posameznem Multiscope

strežniku. Uporaba slednje možnosti je mnogo bolj zaželjena predvsem na račun preglednosti. Glede na princip porazdelitve pristojnosti je namreč smiselno (velikokrat celo nujno) določevati parametre posameznega avtonomnega sistema na njem samem, tako da njegova avtonomnost v nobenem primeru ne more biti prizadeta. Na tak način lahko dosežemo trden, na napake čimbolj odporen sistem.

Dosedaj opisane metode so izpeljanje iz izvornega razreda. Da lahko izvedemo komunikacijo, moramo vpeljati še nekatere interne metode, ki so predstavljene v nadaljevanju.

Ko VideoServer sprejme poizkus povezave gonilnika, nam objekt `fClient` proži dogodek `OnConnectedChange`. Nanj je vezana metoda `VSConnectedChange` (izsek 11).

```
procedure TDevice_VideoServer.VSConnectedChange(Sender: TObject;
  const connected: boolean);
begin
  if connected then begin
    fClient.SendString('#0001|1|','');
    fClient.SendString('#0009|1|','');
    fClient.SendString('#0011|1|','');
  end else begin
    NotifyStatChange(rdsError);
  end;
end;
```

Izsek 11: Implementacija metode VSConnectedChange

V metodi `VSConnectedChange` se najprej ugotovi, ali je bila metoda klicana ob povezavi ali ob odklopu. V kolikor je klicana ob vklopu, se na strežnik pošljejo tri zahteve: naročilo na Multiscope dogodke, povpraševanje po seznamu poznanih Multiscope strežnikov ter naročilo na sporočila glede spremembe stanja povezav na Multiscope strežnike. Če je bila metoda klicana ob odklopu strežnika, se spremeni status gonilnika tako, da odraža napako na komunikaciji (`rdsError`).

Po odposlanju zahtev se pričakuje najprej odgovor, ki bo gonilniku sporočil, katere Multiscope strežnike VideoServer pozna in kakšna so trenutna stanja povezav z njimi. Objekt `fClient` proži dogodek `OnMSCListAvailable` takrat, ko mu VideoServer dostavi seznam. Pri tem se proži metoda `VSMSCListAvail` (izsek 12).

```
procedure TDevice_VideoServer.VSMSCListAvail(Sender: TObject;
  const Napaka: boolean; Lista: TList);
```

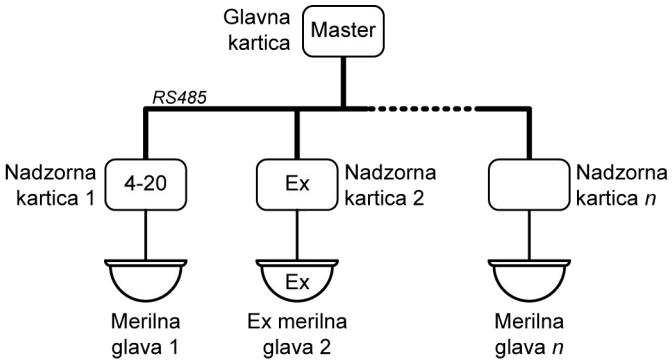
```
var
  i: integer;
  Info: PMSCInfo;
  Elem: TVSElement;

  { funkcija za iskanje MSC-ja med statusi iz VideoServerja }
  function  FindMSCInfoByName(mList: TList; Name: string): PMSCInfo;
  var
    j: integer;

  begin
    result:=nil;
    for j:=0 to mList.Count-1 do begin
      if UpperCase(PMSCInfo(mList[j])^.MSCName) = UpperCase(Name)
      then begin
        result:=mList[j];
        break;
      end;
    end;
  end;

begin
  { to je edini verodostojen način, da ugotovimo, ali je povezava
uspešna }
  NotifyStatChange(rdsRun);

  for i:=0 to fInternalList.Count-1 do begin
    Elem:=fInternalList[i];
    if Elem.Server then begin
      { poglejmo, če server obstaja po novem }
      Info:=FindMSCInfoByName(Lista, Elem.MSCName);
      { če server na novi listi ne obstaja, javi napako }
      if Info = nil then begin
        CreateEvent(Elem.Link.Address,
          cVSDogodki[vsdServerUnknown]);
      end else begin
        if Info.Status <> Elem.Status then begin
          Elem.Status:=Info.Status;
          CreateEvent(Elem.Link.Address,
            cVSDogodki[TVSDogodki(ord(Info^.Status))]);
        end;
      end;
    end;
  end;

  for i:=0 to Lista.Count-1 do begin
    Info:=Lista[i];
    Dispose(Info);
  end;
  Lista.Free;
end;
```

Izsek 12: Implementacija metode VSMSCListAvail

Zaradi narave TCP/IP implementacije v Windows operacijskih sistemih, ki ob neuspešnem priklopu gosta na strežnik javi hkrati priklop in odklop, je smiselno postaviti status naprave v operativno stanje – `rdsRun` – šele takrat, ko sprejmemo seznam Multiscope strežnikov.

4.2 Draeger Regard

Draeger je nemški proizvajalec zaščitne opreme, ki med drugim ponuja tudi sisteme za merjenje koncentracije oziroma zaznavo plinov. Serija Regard [18], za katero je bil gonilnik napisan, je namenjena stalnemu nadzoru vnetljivih, strupenih in drugih plinov ter upravljanju z alarmi, indikatorji in ostali opremi preko alarmnih relejev.



Slika 13: Draeger Regard sistem

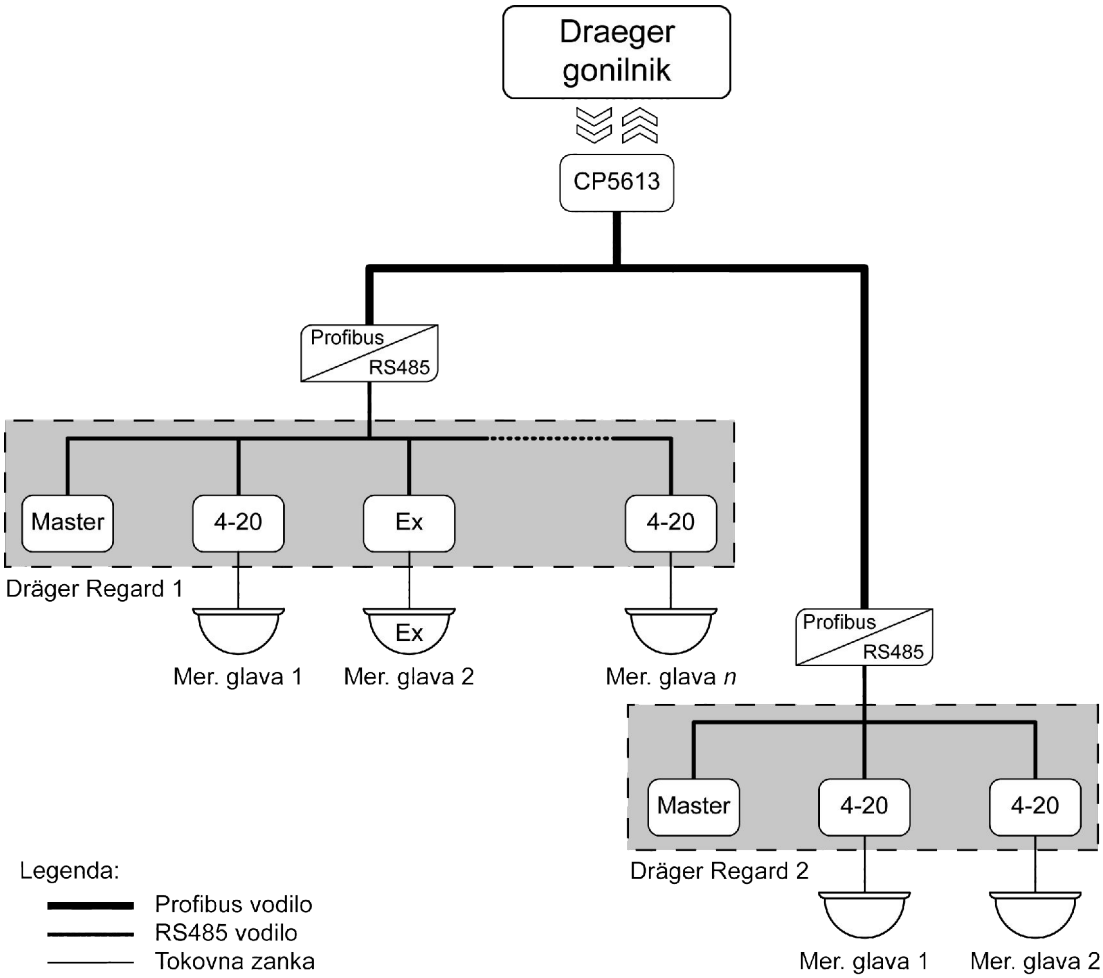
Regard serijo sestavljajo nadzorne kartice (ang. *Control cards*) in glavne kartice (ang. *Master cards*), kot je to prikazano na sliki 13. Na vsako nadzorno kartico lahko priključimo po eno merilno glavo v navadni ali SE-Ex[13] izvedbi. Izmerjeno koncentracijo plina se prenaša tokovno (v navadni izvedbi je to signal med 4 mA in 20 mA).

Glavna kartica diktira komunikacijo na vodilu, obenem pa omogoča proženje skupnih in glasovanih alarmov, skupno ponastavitev (reset) sistema in skupinsko nastavljanje parametrov. Skupni alarm se proži takrat, ko je katerakoli izmed nadzornih kartic v alarmu. Glasovani alarm je termin, ki pomeni proženje alarma takrat, ko je v alarmu *n* izmed *m* kartic, kjer je *m* število nadzornih kartic v podani skupini. Na glavno kartico lahko preko RS485 vodila priključimo največ 63 nadzornih kartic.

Glede na spremembo koncentracije, lahko določimo največ tri alarmne prage (vsak od njih je lahko občutljiv ali na padanje ali naraščanje koncentracije) oziroma največ dva alarmna praga in eno javljanje napake. Napako lahko sproži padec vhodnega

13 Ex: oznaka, ki pomeni, da je naprava izvedena protiekslpozijsko (uporablja se tudi izraz, da je intrinzično varna)

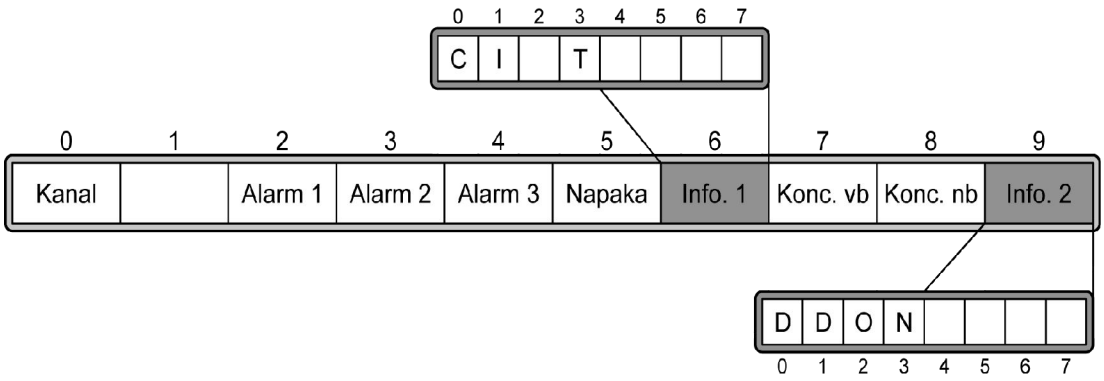
signala pod določeno vrednost, narast signala nad določeno vrednost, kratek stik na napajanju Ex glave ali avtomatsko zaznana napaka pri testiranju glave (če merilna glava to podpira).



Slika 14: Shematski prikaz priključitve Draeger sistema

Na sliki 14 je shematsko prikazan koncept povezave gonilnika na Regard sistem. Na RS485 vodilo vsakega Regard sistema je priključen komunikacijski pretvornik RS485/profibus. Draeger v ta namen ponuja izdelek *Unigate* proizvajalca Deutschmann Automation, GmbH, Bad Camberg. V osnovi je to splošnonamenski pretvornik iz vodil RS232, RS422 ali RS485 na profibus vodilo prilagojen tako, da pravilno interpretira Draegerjev interni protokol. Na tak način lahko povežemo večje število Regard sistemov preko enega profibus vodila, saj ima vsak pretvornik svojo identifikacijsko številko (številko profibus vozla).

Za komunikacijo z napravami na profibus vodilu uporablja gonilnik vmesnik Siemens CP5613 v DP načinu. Ta je hkrati tudi glavna postaja (ang. *Master station*). Regard nadzorne kartice so v nadaljevanju poimenovane s splošnejšim pojmom – kanali. Ta termin se uporablja v dokumentaciji protokola in se zaradi konsistence uporablja tudi v pričujočem delu.



Slika 15: Struktura vhodnih profibus podatkov

Na sliki 15 je prikazana struktura vhodnih podatkov, ki jih gonilnik pridobi preko profibus vmesnika. Poimenovani so le podatki, ki jih gonilnik uporablja. Vsako od poimenovanih polj zaseda 1 zlog, kar da skupno dolžino podatkovnega bloka 10 bajtov. Na sliki je uporabljeno številčenje bajtov in bitov, ki se zaradi lažjega sledenja izvorne kode začne z 0. Tako številčenje je pogosto v računalništvu, ker nudi precej prednosti pred številčenjem, ki se začne z ena in je uporabljeno tudi v gonilniku.

Prvi zlog (na sliki označen z 0) nosi informacijo, na kateri kanal se nanašajo ostali podatki. Pomen preostalih bajtov je naslednji:

- Alarm 1, Alarm 2, Alarm 3: stanje posameznega alarmnega praga; upoštevata se prvi in tretji bit, njihov pomen je naslednji (x pomeni, da vrednost na tistem mestu ni pomembna):
  - $0x0_2$: alarmni prag trenutno ni dosežen,
  - $0x1_2$: alarmni prag trenutno ni dosežen, je pa bil dosežen in takratni alarm še ni potrjen,
  - $1x0_2$: alarmni prag je dosežen in velja alarmno stanje, operater je alarm potrdil,

  - $1x1_2$: alarmni prag je dosežen in velja alarmno stanje, operater alarma še ni potrdil.
- Napaka: status napake na kanalu; upoštevata se prvi in tretji bit, njihov pomen je naslednji (x pomeni, da vrednost na tistem mestu ni pomembna):
  - $0x0_2$: trenutno ni napake,
  - $0x1_2$: trenutno ni napake, vendar pretekla napaka še ni potrjena,
  - $1x0_2$: zaznana je napaka na kanalu, operater jo je potrdil,
  - $1x1_2$: zaznana je napaka na kanalu, operater jo še ni potrdil.
- Info. 1, Info. 2: polji, imenovani tudi zastavni (ang. *Flag fields*), ki nosita informacije o določenih stanjih sistema; s črkami so označeni biti, ki pomenijo naslednje:
  - C: če ima bit vrednost 1, je merilna glava v postopku kalibracije;
  - I: če ima bit vrednost 1, je kanal v stanju, ko se morebitne alarme in napake ne sme upoštevati. To se lahko zgodi v primeru inicializacije kanala (pri zagonu) ali ko je kanal v stanju konfiguracije;
  - T: če ima bit vrednost 1, je potekel čas, v katerem se mora kanal javiti glavni kartici. Ta bit nastavi Unigate pretvornik, če že dalj časa ni zasledil komunikacije s kanalom oziroma se kanal ne odziva več;
  - D: število decimalnih mest podane koncentracije. Potrebno je upoštevati skupno vrednost obeh bitov, označenih z D;
  - O: če ima bit vrednost 1, merilna glava sporoča vrednost izven pričakovanega območja. To navadno pomeni napako glave ali okvaro kabelskih vodov do glave;
  - N: predznak izmerjene koncentracije. Vrednost 1 pomeni negativen predznak, vrednost 0 pozitiven.
- Konc. vb, Konc. nb.: bajta, ki podajata vrednost izmerjene koncentracije. Najprej je podan visoki zlog (označen z vb), nato nizki zlog (označen z nb).

Ker lahko naenkrat pridobimo informacije le o enem kanalu, moramo Unigate pretvorniku sporočiti, kateri kanal nas zanima. V ta namen je definirano izhodno sporočilo, dolgo 3 bajte, ki v prvem bajtu nosi številko željenega kanala.

Razred gonilnika se imenuje `TDevice_Draeger` in je izpeljan iz razreda `TProfibusDeviceDriver`. Definicija razreda je razvidna z izseka 13.

```
  TDraegerElement = class(TObject)
  private
    fAddress: integer;
    fChannel: integer;
    fGas: real;
    fEvent: string;
    fLink: TDevElement;
    procedure SetLink(const Value: TDevElement);
  public
    property Link: TDevElement read fLink write SetLink;
    property Address: integer read fAddress;
    property Channel: integer read fChannel;
    property Gas: real read fGas write fGas;
    property LastEvent: string read fEvent write fEvent;
  end;

  TDevice_Draeger = class(TProfibusDeviceDriver)
  private
    fInternalList: TList;
    fOutBuf: TDataArray;
    function Add(_Link: TDevElement): TDraegerElement;
    procedure Clear;

    function AnalyseEvent(Data: TDataArray; Ch: TDraegerElement):
byte;
    procedure ReportGasValue(Adresa, ChNo: integer; GasVal: string);
  public
    constructor Create; reintroduce;
    destructor Destroy; override;

    procedure Cycle; override;
    function GetDriverVersion: TModulVersion; override;
    function GetDriverInfo: string; override;
    procedure Initialize(_Link: TRemDev); override;
    procedure Finalize; override;
    function Stop(const timeout: Integer = 5000): Boolean; override;
  end;
```

Izsek 13: Definicija razreda TDevice_Draeger in pomožnega razreda TDraegerElement

Gonilnik implementira konstruktor `Create`, kjer poleg podedovanega obnašanja poskrbi še za inicializacijo internih spremenljivk. Destruktor `Destroy` sprosti pomnilnik, zavzet v konstruktorju, nato kliče svojo predhodno implementacijo.

Metodi `GetDriverInfo` in `GetDriverVersion` vračata opis in verzijo izvedbe gonilnika. Funkcionalno ne vplivata na delovanje gonilnika, pomagata pa razločevati posamezne gonilnike in zagotavljata sledljivost verzij.

```
procedure TDevice_Draeger.Initialize(_Link: TRemDev);
var
  i: integer;
```

```
begin
  inherited;
  Clear;
  for i:=0 to _Link.Elements.Count-1 do begin
    Add(_Link.Elements[i]);
  end;
end;
```

Izsek 14: Implementacija metode Initialize v gonilniku Draeger

```
function TDevice_Draeger.Add(_Link: TDevElement): TDraegerElement;
begin
  result:=TDraegerElement.Create;
  result.Link:=_Link;
  if (not (result.Channel in [1..99])) or
    (not (result.Address in [0..127]))
  then begin
    CreateEvent(_Link.Address, 'CHERROR');
    result.Free;
    result:=nil;
  end else
    fInternalList.Add(result);
end;
```

Izsek 15: Implementacija metode Add v gonilniku Draeger

Implementacija metode `Initialize` (izsek 14) poskrbi za prepis podatkovne strukture v interno predstavitev (konkretno v objekte tipa `TDraegerElement`), ki olajša analiziranje podatkovnih paketov. Prepisovanje je izvedeno z metodo `Add` (izsek 15), ki hkrati tudi preveri pravilnost podatkov. V kolikor je številka kanala ali številka profibus vozlja izven pričakovanega območja ali je naslov elementa popolnoma napačen, se na tem elementu sproži dogodek `CHERROR`. Element, na katerem se tako napako odkrije, ni vnešen v notranji seznam elementov, s čimer se poenostavi tudi notranja zanka komunikacije. V njej namreč ni potrebno dodatno preverjati pravilnosti parametrov elementov, ki so nujni za pridobivanje njihovih podatkov.

```
procedure TDevice_Draeger.Clear;
var
  i: integer;

begin
  for i:=0 to fInternalList.Count-1 do begin
    TDraegerElement(fInternalList[i]).Free;
  end;
  fInternalList.Clear;
end;
```

Izsek 16: Implementacija metode Clear v gonilniku Draeger

Metoda `Finalize` sprosti tiste podatke, ki so bili zajeti ob klicu metode `Initialize`, torej v tem primeru interni zapis podatkov. To stori s klicem metode

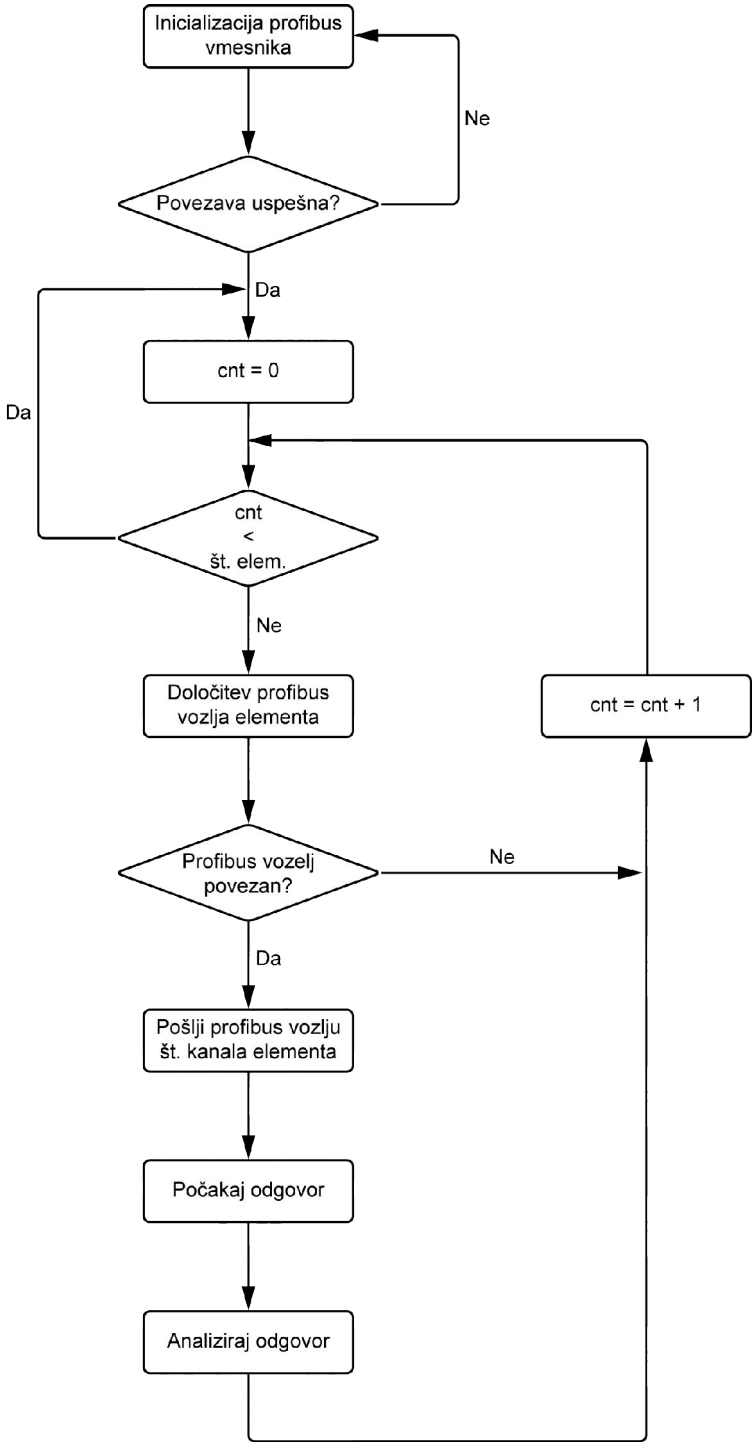
`Clear` (izsek 16), ki poskrbi za sprostitev vseh notranjih elementov in njihovih kazalcev.

```
function TDevice_Draeger.Stop(const timeout: Integer): Boolean;
var
  pbError: TProfiBusError;
  cnt: integer;
  pbTimeOut: integer;
  i: integer;

begin
  result:=inherited Stop(Timeout);
  NotifyInfo('Going down...', '');
  cnt:=2;
  while ProfiBus.Connected do begin
    if not ProfiBus.Disconnect(pbError) then begin
      pbTimeOut:=2;
      NotifyInfo('Error disconnecting: repeat in ' +
        IntToStr(pbTimeOut) + 's', '');
      while pbTimeOut > 0 do begin
        NotifyInfo(IntToStr(pbTimeOut)+'s', '');
        sleep(1000);
        dec(pbTimeOut);
      end;
      dec(cnt);
      if cnt<=0 then
        break;
    end;
  end;
  if not ProfiBus.Connected then begin
    NotifyInfo('Disconnected', '');
  end else begin
    NotifyStatChange(rdsFatalError);
    NotifyInfo('Error while trying to disconnect after aprox. 2
sec', '');
  end;
  for i:=0 to fInternalList.Count-1 do begin
    TDraegerElement(fInternalList[i]).Gas:=1e-7;
    TDraegerElement(fInternalList[i]).LastEvent:='';
  end;
end;
```

Izsek 17: Implementacija metode Stop gonilnika Draeger

Metoda `Stop` poskrbi za zaustavitev komunikacije (izsek 17). Od svoje predhodnice podeduje ustavljanje komunikacijske niti, nato pa, v kolikor je povezava vzpostavljena, zahteva odklop gonilnika. Če pride do napake pri odklopu (v primeru, da vmesnik še ni poslal vseh podatkov), se sproži ponovitveni mehanizem. Če po dveh ponovitvah odklop še vedno ni uspešno izveden, se dodeli gonilniku status kritične napake (`rdsFatalError`). Pojav tega stanja navadno pomeni napako na programski in/ali strojni opremi, zaradi česar je priporočljivo izvesti servisni pregled.

Celotna komunikacija je izvedena v `Cycle` metodi (izsek 18). Diagram poteka, na katerem temelji njeno delovanje, je prikazan na sliki 16.



Slika 16: Diagram poteka za metodo Cycle gonilnika Draeger

```
procedure TDevice_Draeger.Cycle;
var
```

```
  pbError: TProfiBusError;
  cnt: integer;
  Napaka: boolean;
  Ch: TDraegerElement;
  InBuf: TDataArray;

begin
  if not ProfiBus.Connected then begin
    NotifyInfo('Trying to connect to CP5613...',
      ProfiBus.AccessPoint);
    if not ProfiBus.Connect(pbError) then begin
      NotifyInfo('Error on first run with message:',
        pbError.OpisNapake);
      NotifyInfo('Trying to reset device ...', '');
      if ProfiBus.ResetDevice(pbError) then begin
        NotifyInfo('  reset OK, trying to reconnect ...', '');
        if not ProfiBus.Connect(pbError) then begin
          NotifyInfo('Error on reconnect with message: ',
            pbError.OpisNapake);
        end;
      end else
        NotifyInfo('Error on reseting device with message: ',
          pbError.OpisNapake);
    end else
      NotifyInfo('  Connection OK', '');
    FirstCycle:=true;
    ReportGasValue(-1, -1, '???');
  end;
  begin
    if FirstCycle then begin
      NotifyInfo('Initialising Send Buffer...', '');
      fOutBuf[1]:=0;
      fOutBuf[2]:=0;
      NotifyInfo('Running Main Loop with 200ms Delay...', '');
      Sleep(200);
    end else begin
      NotifyInfo('Main loop running...', '');
      cnt:=0;
      Napaka:=false;
      { Draeger ima kanale od 1 do 99 }
      while (cnt < fInternalList.Count) do begin
        Ch:=fInternalList[cnt];
        ProfiBus.SlaveAddress:=Ch.Address;
        if ProfiBus.SlaveAlive then begin
          fOutBuf[0]:=Ch.Channel;
          if ProfiBus.WriteData(fOutBuf, pbError) then begin
            sleep(200);
            if ProfiBus.ReadData(InBuf, pbError) then begin
              if AnalyseEvent(InBuf, Ch) = 1 then begin
                fOutBuf[1]:=1;
                { poresetiramo status na tem kanalu }
                ProfiBus.WriteData(fOutBuf, pbError);
                fOutBuf[1]:=0;
              end;
            end else begin
              NotifyInfo('Error reading data with message: ',
                pbError.OpisNapake);
              Napaka:=true;
              break;
            end;
            SetLength(InBuf, 0);
```

```
          end else begin
            NotifyInfo('Error writing data with message: ',
              pbError.OpisNapake);
            Napaka:=true;
            break;
          end;
        end else begin
          Napaka:=true;
          ReportGasValue(Ch.Address, Ch.Channel, '***');
        end;
        inc(cnt);
      end;

      if Napaka then
        NotifyStatChange(rdsError)
      else
        NotifyStatChange(rdsRun);
      Sleep(250);
    end;
  end;
end;
```

Izsek 18: Implementacija metode Cycle gonilnika Draeger

Inicializacija komunikacijskega vmesnika je obvezna. Vmesnik CP5613 se nastavi glede na podano dostopno točko (ang. *Access point*). Glede na dostopno točko se naloži pripadajoča konfiguracijska datoteka, ki opisuje parametre vodila ter število in tip vozljev. Ko je inicializacija uspešno izvedena, se začne ciklično poizvedovanje o stanju na posameznih kanalih. Po vsaki poslani zahtevi o stanju kanala je potrebno počakati določen čas, ki ga potrebuje podatek, da je dostavljen naslovniku. Ta nato pripravi zahtevane podatke in jih vrne pošiljatelju.

```
function TDevice_Draeger.AnalyseEvent(Data: TDataArray;
  Ch: TDraegerElement): byte;

type
  TDraegerAlarmStatus = (dasOff, dasUnacknowledged,
dasAcknowledged);

const DecimalPlaces: array[0..3] of real = (1, 0.1, 0.01, 0.001);

var
  Gas: real;
  EventStr: string;
  Address: string;

  function ByteToAlarm(const b: byte): TDraegerAlarmStatus;
  begin
    if b in [0, 2] then
      result:=dasOff
    else if b in [1, 3, 5, 7] then
      result:=dasUnacknowledged
    else
      result:=dasAcknowledged;
  end;
```

```
begin
  result:=0;
  if high(Data)<9 then begin
    NotifyInfo('Napaka pri obdelavi: dolzina manjsa od 9', '');
    exit;
  end;
  Gas:=(integer(Data[7] SHL 8) OR Data[8]) *
    DecimalPlaces[(Data[9] AND 3)];
  if (Data[9] AND 8)>0 then
    Gas:=-Gas;

  EventStr:='';

  if (Data[6] AND 8) > 0 then
    EventStr:='TIMEOUT'
  else if (Data[6] AND 1) > 0 then
    EventStr:='HEADCALIB'
  else if ByteToAlarm(Data[5]) = dasUnacknowledged then
    EventStr:='FAULTUNACK'
  else if ByteToAlarm(Data[5]) = dasAcknowledged then
    EventStr:='FAULTACTIVE'
  else if ByteToAlarm(Data[4]) = dasUnacknowledged then
    EventStr:='A3UNACK'
  else if ByteToAlarm(Data[4]) = dasAcknowledged then
    EventStr:='A3ACTIVE'
  else if ByteToAlarm(Data[3]) = dasUnacknowledged then
    EventStr:='A2UNACK'
  else if ByteToAlarm(Data[3]) = dasAcknowledged then
    EventStr:='A2ACTIVE'
  else if ByteToAlarm(Data[2]) = dasUnacknowledged then
    EventStr:='A1UNACK'
  else if ByteToAlarm(Data[2]) = dasAcknowledged then
    EventStr:='A1ACTIVE'
  else if (Data[9] AND 4) > 0 then
    EventStr:='OUTOFRANGE'
  else if (Data[6] AND 2) > 0 then
    EventStr:='ALINHIBIT'
  else
    EventStr:='CHOK';

  if (Ch.LastEvent <> EventStr) then begin
    { sprememba v statusu }
    Address:=Format('A%dCH%d', [ch.Address, Data[0]]);
    NotifyInfo(Format('Got new event for %s', [Address]), EventStr);
    Ch.LastEvent:=EventStr;
    Ch.Gas:=Gas;
    CreateEvent(Address, EventStr, 0, '', '',
      FloatToStrF(Gas, ffFixed, 6, (Data[9] AND 3)));
    result:=1;
  end else if Ch.Gas <> Gas then begin
    Ch.Gas:=Gas;
    Address:=Format('A%dCH%d', [ch.Address, Data[0]]);
    NotifyInfo('Reporting Gas Value for Address:', Address);
    CreateEvent(Address, '', 0, '', '',
      FloatToStrF(Gas, ffFixed, 6, (Data[9] AND 3)));
  end;
end;
```

Izsek 19: Implementacija metode AnalyseEvent gonilnika Draeger

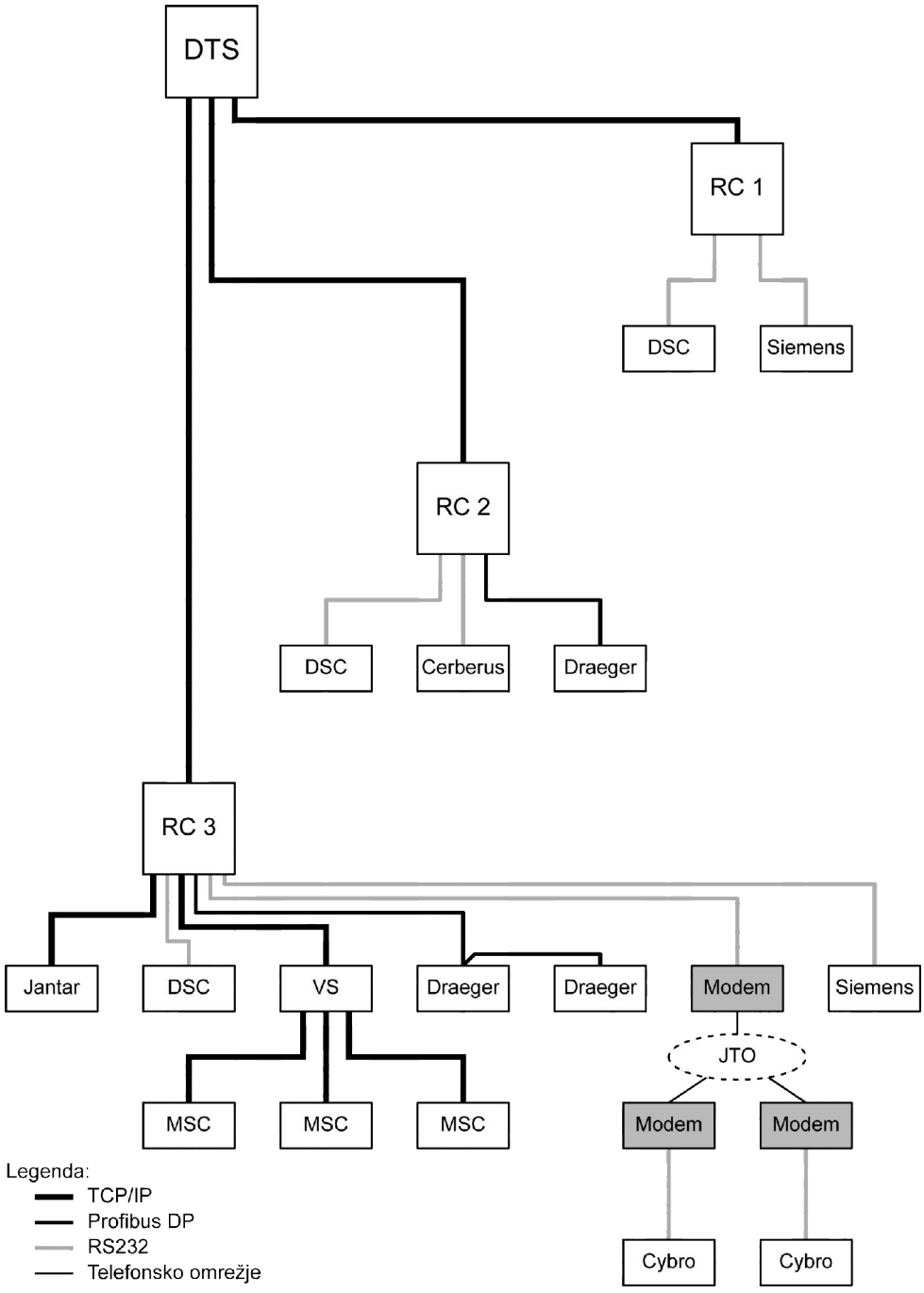
Analiza podatkov se opravlja v metodi `AnalyseEvent` (izsek 19). Glede na prej

opisano strukturo podatkov se iz podane podatkovne vrste `Data` pridobi trenutno stanje kanala. Vsakemu stanju je prirejen en dogodek naprave. Dogodki se določajo prioritetno z naslednjimi stopnjami (od najvišje prioritete proti najmanjši):

1. `TIMEOUT`: kanal se ne odziva, morebiti navedena stanja niso več verodostojna;
2. `HEADCALIB`: merilna glava je v kalibraciji, morebitna stanja so posledica kalibracije glave;
3. `FAULTUNACK`: stanje nepotrjene napake, na kanalu je bila zaznana napaka, ki je operater še ni potrdil;
4. `FAULTACTIVE`: stanje potrjene napake, kanal zaznava napako, ki jo je operater že potrdil, ni pa še bila odpravljena;
5. `A3UNACK`: alarmno stanje tretje stopnje, presežen je bil tretji prag koncentracije, alarma operater še ni potrdil;
6. `A3ACTIVE`: alarmno stanje tretje stopnje, tretji prag koncentracije je še vedno presežen, operater je potrdil alarmno stanje;
7. `A2UNACK`: alarmno stanje druge stopnje, presežen je bil drugi prag koncentracije, alarma operater še ni potrdil;
8. `A2ACTIVE`: alarmno stanje druge stopnje, drugi prag koncentracije je še vedno presežen, operater je potrdil alarmno stanje;
9. `A1UNACK`: alarmno stanje prve stopnje, presežen je bil prvi prag koncentracije, alarma operater še ni potrdil;
10. `A1ACTIVE`: alarmno stanje prve stopnje, prvi prag koncentracije je še vedno presežen, operater je potrdil alarmno stanje;
11. `OUTOFRANGE`: stanje napake signala, vrednost signala koncentracije iz merilne glave je izven pričakovanega območja;
12. `ALINHIBIT`: stanje blokade alarmov, prisotni alarmi so lažni;
13. `CHOK`: normalno stanje kanala, koncentracija plina ne presega podanih pragov, glava deluje normalno, drugih napak ni.

V kolikor pride do spremembe stanja na kanalu, se to propagira kot proženje dogodka na pripadajočem elementu. Če pride do spremembe koncentracije plina, se to proži kot nemi dogodek, ki nosi samo informaciji o naslovu elementa in aktualni koncentraciji.

5. Primer realizacije porazdeljenega sistema za zajem podatkov



Slika 17: Praktičen primer izgleda porazdeljenega sistema za zajem podatkov

Na sliki 17 je prikazan praktičen primer realizacije porazdeljenega sistema za zajem podatkov. Uporabljene so naslednje kratice:

- DTS: osrednji strežnik in koordinator (iz ang. *Data Transfer Server*),
- RC *x*: oddaljeni komunikator (iz ang. *Remote Communicator*),
- VS: Video Server,
- JTO: javno telefonsko omrežje.

Naprave različnih proizvajalcev so priključene na posamezni oddaljeni komunikator, kjer za interakcijo z njimi skrbijo gonilniki, izpeljani iz predstavljenega generičnega razreda. Vidimo lahko, da so za komunikacijo uporabljeni kar štirje različni prenosni mediji: TCP/IP, Profibus, RS232 in telefonsko omrežje.

Jantar je slovenski proizvajalec sistemov za kontrolo pristopa. Za komunikacijo s temi napravami je uporabljen protokol TCP/IP v gostujočem načinu (ang. *TCP Client*). Naprave omogočajo posredovanje dogodkov iz sistema na priključene goste – v tem primeru gonilniku oddaljenega komunikatorja – in izvrševanje ukazov, ki jih od gostov prejmejo.

DSC je kanadski proizvajalec protivlomnih sistemov. Komunikacija z njihovimi protivlomnimi centralami poteka preko RS232 vodila. Komunikacijski protokol omogoča zajem podatkov o alarmnih stanjih in morebitnih napakah ter vklop in izklop varovanja posameznih segmentov.

Video Server je uporabljen kot koncentrator za tri Geutebrueck Multiscope strežnike. Ti mu pošiljajo sporočila o zaznavi premikov in izpadih signalov video kamer, ki jih Video Server posreduje gonilniku v oddaljenemu komunikatorju.

V sistem so vključeni trije neodvisni Draeger Regard sistemi. Dva od teh sta priključena na isto Profibus vodilo. RC 2 in RC 3 imata nameščeni glavni Profibus kartici Siemens CP5613, kateri preko Profibus DP-V0 protokola ciklično zajemata podatke iz priključenih Regard sistemov.

V sistem so vključene še Siemens protipožarne centrale in Cerberus kombinirane protipožarne in protivlomne centrale. Komunikacija s temi centralami poteka preko RS232 vodila in je enosmerna – mogoče je samo zajemati podatke, upravljati z napravo preko komunikacijskega protokola ni mogoče.

Preko telefonske linije se v sistem občasno javljata dva Cybro programabilna logična

krmilnika. Vsak izmed njiju opravlja nadzor oddaljenega objekta in ima predpisan čas, v katerem se mora javiti gonilniku, da ga ta šteje za delujočega. V primeru, da krmilnik zazna alarmno stanje začne nemudoma vzpostavljati telefonsko povezavo z gonilnikom.

Razdalje med DTS in RC so različne; najbližji je oddaljen le nekaj metrov, do najbolj oddaljenega pa je okoli 3 km. Povezava je izvedena preko obstoječega intranet omrežja, ki ga podjetje uporablja tudi za pisarniške povezave. Izmerjena odzivnost v času največje obremenitve je bila boljša od 1 sekunde. Dodatni varnostni mehanizmi v komunikacijskem protokolu med DTS in RC omogočajo visoko odpornost sistema na izpade mreže ali morebitne motnje v prenosu.

6. Zaključek

V diplomski nalogi je predstavljeno ogrodje generičnega gonilnika, ki omogoča izvedbo poljubnega gonilnika za komunikacijo z napravami tehničnega varovanja. Povezan v paket Integra Security omogoča učinkovito informatizacijo tovrstnih naprav, pri čemer so glavne prednosti v primerjavi s konkurenčnimi produkti naslednje:

- porazdeljen način zajema in obdelave podatkov omogoča razporeditev obremenitve na več cenenih osebnih računalnikov;
- posamezni členi sistema med seboj komunicirajo preko standardiziranega in široko dostopnega protokola TCP/IP, kar pomeni, da je za povezavo moč uporabiti raznovrsten nabor komunikacijskih vmesnikov in medijev;
- neodvisnost od proizvajalcev naprav tehničnega varovanja;
- kratek razvojni čas novega gonilnika;
- centralizirano upravljanje sistema;
- za vizualizacijo stanja priključenih naprav je uporabljena vektorska grafika, ki omogoča dinamično povečevanje slike z možnostjo skrivanja določenih elementov;
- možnost prikaza žive video slike iz večih kamer hkrati ali arhivskega posnetka;

Kot alternativo predlagani rešitvi bi lahko predlagali uporabo OPC tehnologije. Vendar se moramo pri tem zavedati nekaterih pomankljivosti, ki jih ta prinaša. OPC tehnologija ne omogoča centraliziranega upravljanja večih strežnikov, deluje na COM/DCOM[14] tehnologiji, ki je bila v zadnjem času glavna tarča napadov na računalnike, in je na voljo le v Windows okolju. Novejši OPC strežniki za komunikacijo uporabljajo XML[15] standard, ki deluje tudi v Unix/Linux okolju, vendar je takih strežnikov zaenkrat zelo malo.

Kot je že bilo omenjeno, za večino naprav tehničnega varovanja OPC strežniki ne obstajajo. Predlagano ogrodje generičnega gonilnika lahko uporabimo tudi za pisanje

14 COM: Component Object Model
DCOM: Distributed Component Object Model
15 XML: Extensible Markup Language

tistega dela OPC strežnika, ki z napravo komunicira. Groba zasnova celotnega člena bi se sicer spremenila, vendar princip komunikacije z napravami ostane enak.

Literatura

[1] David Bailey, *Practical SCADA for industry*, Newnes, Oxford, UK, 2003

[2] Axel Daneels, Wayne Salter, *What is SCADA?*, 2000, http://ref.web.cern.ch/ref/CERN/CNL/2000/003/scada

[3] OPC Foundation, *OPC Specifications*, 2005, http://www.opcfoundation.org/Downloads.aspx?CM=1&CN=KEY&CI=283

[4] William Buchanan, *Applied data communications and networks*, Chapman & Hall, London, UK, 1996

[5] Lammert Bies, *RS232 Specifications and standard*, 2005, http://www.lammertbies.nl/comm/info/RS-232_specs.html

[6] W. Richard Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley publishing company, Reading, MA, 1994

[7] USB Implementers Forum, *USB 2.0 Specification*, 2000, www.usb.org/developers/docs

[8] 1394 Trade Association, *1394 Technology*, 2005, http://www.1394ta.org/Technology/

[9] Richard Zurawski, *The Industrial Communication Technology Handbook*, CRC Press, Boca Raton, Florida, 2005

[10] dr. Peter Šuhel, dr. Boštjan Murovec, *Računalniška integracija proizvodnje*, Gorenje, d. d. Izobraževalni center, Velenje, 2003

[11] Bruce Schneier, *Applied Cryptogprahy, Second Edition*, John Wiley & Sons, Oak Park, Illinois, 1996

[12] Edward N. Dekker, *Developing Windows NT Device Drivers*, Addison-Wesley, Reading, Massachusetts, 1999

[13] Steve Teixeira, Xavier Pacheco, *Borland Delphi 6 Developer's guide*, Sams Publishing, Indianapolis, IN, 2002

[14] John Ayres, *Tomes of Delphi: Win32 Core API Windows 2000 Edition*, Wordware Publishing, Plano, TX, 2002

[15] Francois Piette, *Overbyte products - ICS*, 2005, http://www.overbyte.be/frame_index.html

[16] David A. Solomon, *Inside Windows 2000, 3rd ed.*, Microsoft Press, Redmond, Washington,

[17] Geutebrueck GmbH, *Schnittstellen der Videosicherheitssysteme*, 2005, http://www.geutebrueck.de/de/schnittstellen_sdk/schnittstellen/schnittstellen.php

[18] Draeger Safety, *Regard Control System*, 2005, http://www.draeger.com/ST/internet/SI/en/Industries/GDS/General/Controllers/ModularRack/ggi_regard_rack_system.jsp

Izjava

Izjavljam, da sem diplomsko delo izdelal samostojno pod vodstvom mentorja doc. dr. Boštjana Murovca, univ. dipl. inž. el. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

V Ljubljani, 1. septembra 2005 Marko Ušaj