České vysoké učení technické v Praze

Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE

Vývoj zařízení Profinet IO

Pavel Mezera

Katedra řídicí techniky

Praha 2008

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky *Školní rok:* 2005/2006

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Pavel Mezera

Obor: Technická kybernetika

Název tématu: Vývoj zařízení Profinet IO

Zásady pro vypracování:

1. Seznamte se s protokolem Profinet IO a s vývojovou deskou NetSilicon Net+50.
2. Implementujte stávající firmware k Profinet IO device pod operačním systémem ThreadX na této desce.
3. Vyberte vhodnou HW platformu a vhodný operační systém reálného času a převeďte sem implementovaný a otestovaný software pro Profinet IO device.

Seznam odborné literatury:

[1] Manfred Popp, Karl Weber: The Rapid Way to Profinet. Profibus International, Karlsruhe. 2004.
[2] Profibus International: Profinet Specification. Profibus International, Karlsruhe. 2004.

Vedoucí diplomové práce: Ing. Pavel Burget

Termín zadání diplomové práce: zimní semestr 2005/2006

Termín odevzdání diplomové práce: leden 2007

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

prof. Ing. Vladimír Kučera, DrSc.
děkan

V Praze dne 20.01.2006

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze, dne 18. ledna 2008

Pavel Meszen
.....
podpis

Poděkování

Především bych chtěl na tomto místě poděkovat vedoucímu mé diplomové práce Ing. Pavlu Burgetovi za jeho podporu a cenné rady které přispěly ke vzniku této práce.

Mé poděkování taktéž patří mojí rodině za podporu při mém dosavadním studiu.

Abstrakt

Tato práce je zaměřena na vývoj Profinet IO zařízení na základě dodaného komunikačního stacku fy. Siemens pro dva různé typy operačních systémů. Pro ThreadX jako zástupce specifických realtime OS a Linux s jádrem 2.6.12 jako zástupce všeobecných OS. Pro tyto dva OS byl komunikační stack upraven a byla pod nimi zařízení zprovozněna. Přenášením vyvolané úpravy komunikačního stacku jsou v této práci popsány stejně jako teoretický úvod do Referenčního modelu OSI a na něm založené specifikace standardu Profinet IO.

Abstract

This thesis is concerned on Profinet IO device development on basis of the supplied communications stack from Siemens Company for two distinct types of OS. For ThreadX which represents specific real-time OS and Linux like representant of general purpose OS. For this two chosen OS was communication stack modified. All induced changes are described in this thesis. Basic description of OSI Reference model and on it based Profinet IO specification is in this thesis too.

Obsah

Prohlášení ..... **Chyba! Záložka není definována.**
Poděkování v
Abstrakt vii
Abstract vii
Obsah viii
Seznam Obrázků x
Úvod 1
Trendy ve vývoji 1
Komunikační stack 1
Operační systémy 3
Analýza problému a definice úloh 4
Přenášení aplikací 4
Implementace 4
ThreadX 5
Linux 5
ISO-OSI Referenční model 7
Historie standardů ISO-OSI 7
Obecný model 7
OSI služby 14
Struktura aplikační vrstvy 17
Profinet IO 19
Standardizace 19
Koncept a vztah k ISO-OSI modelu 19
Profinet IO 20
Typy zařízení 20
IO kontrolér 20
IO supervizor 21
IO parametrizační server 21
IO zařízení 21
PROFINET IO komunikační model 22
Aplikační servisní elementy (ASE) 22
Stavové automaty protokolů ASE 24
Stavové automaty protokolů FAL 24
Remote procedure call 28
GSDML 29
Profinet IO stack 31
Vývoj pro ThreadX 35
Vývojová deska NetSilicon Net+50 35
Velikost písmen v názvech souborů 37
Zarovnávání struktur 37
Úprava definicí v compiler.h 39
Úprava stacku vhledem k BSP NetOS 6.3 39
Vývoj pro Linux 41
Vývojová deska FS Forth-Systeme A9M9750DEV 41
Linux 41
Překlad aplikace 41
Překlad modulu 42
Nová architektura 43

Rozhraní aplikace - jádro 43
Implementace rozhraní 46
Rozhraní jádro – OS, Eth, Sock 48
Instalace PNIO stacku 49
Závěr 51
Literatura: 53

Seznam Obrázků

Obr. 1, Možnosti využití komunikačního procesoru fy. Hilscher 2
Obr. 2, Pohled na reálný systém skrze vrstvy 8
Obr. 3, Systémy, vrstvy a služby otevřeného systému 8
Obr. 4, Uspořádání entit, SAPů a protokolů v RM 9
Obr. 5, Spojení a koncové body spojení (CEPy) 10
Obr. 6, Pojmenování, adresy a CEP identifikátor 10
Obr. 7, Příklad (N) adresáře 10
Obr. 8, Mapování mezi adresami 11
Obr. 9, Korespondence mezi spojeními 11
Obr. 10, Logické vztahy mezi datovými pakety v přiléhajících vrstvách 12
Obr. 11, Reprezentace funkcí pro správu 13
Obr. 12, OSI Model služeb 15
Obr. 13, Příklad časová sekvence pro nepotvrzovanou a potvrzovanou variantu 15
Obr. 14, Příklad definice servisní entity 16
Obr. 15, Příklad potvrzované služby mezi dvěmi uživateli 16
Obr. 16, Příklad nepotvrzované služby mezi dvěma uživateli 16
Obr. 17, Struktura Aplikační entity 17
Obr. 18, Přístup k reálnímu objektu skrze APO 18
Obr. 19, Přechod od 7mi vrstev u OSI ke 3 u průmyslových sítí 20
Obr. 20, Přehled struktury ASE stavových automatů 24
Obr. 21, Přehled struktury FAL stavových automatů 24
Obr. 22, Přehled struktury AR stavových automatů 26
Obr. 23, Přehled DM stavových automatů 27
Obr. 24, Vzdálené volání procedur 28
Obr. 25, Struktura komunikačního stacku 31
Obr. 25, Příklad realizace evidenční struktury virtuálního rozhraní 33
Obr. 26, Princip fungování obslužného vlákna u událostmi řízené aplikace 34
Obr. 27, Příklad realizace obslužného vlákna u událostmi řízené aplikace 34
Obr. 28, Příklad generického makefile využívajícího skripty projektu MakeDep 37
Obr. 29, Příklad nezarovnané struktury 38
Obr. 30, Příklad implicitně zarovnané struktury 38
Obr. 31, Příklad implicitně zarovnané struktury 38
Obr. 32, Příklad nuceně zarovnané struktury u GCC 38
Obr. 33, Příklad nuceně zarovnané struktury u MS překladačů 38
Obr. 34, Příklad generického makefile pro překlad modulu 42
Obr. 35, Nové uspořádání komunikačního software 43
Obr. 36, Princip vzniku a fungování rozhraní 48
Obr. 37, Místo odchytu a odesílání Eth. rámců v relaci s TCP/IP stackem 49
Obr. 38, Sesazení upravených rozhraní a dodaného stacku do výsledné konfigurace 50

Úvod

Dynamický rozvoj v oblasti výpočetní techniky přináší stále nové možnosti řešení, jež byly dříve buď neekonomické nebo nebyly vůbec reálné. Dnešní svět spěje ke konvergenci technologií a oblast výpočetní techniky a její prorůstání do ostatních oblastí je nezpochybnitelné a všudypřítomné. Vždyť dnešní sdělovací technika začíná být pouze aplikací technologií digitálních počítačových sítí. A tak tomu je dnes i u automatizační techniky, která ve svých počátcích reléových řídicích automatů stvořila první počítač. Hlavním jmenovatelem tohoto vývoje jsou náklady a tak, kde je to možné se přejímá již hotové a ozkoušené řešení, proto jsou dnes již programovatelné řídicí automaty konstruovány na běžných standardních procesorech a komunikační sítě kopírují s jistou prodlevou na osvědčení i technologie počítačových sítí.

Tímto směrem jde i Profinet IO, který vychází koncepčně z Profibusu DP postaveného na standardní fyzické vrstvě RS-485 avšak tuto dnes již zastaralou technologii nahrazuje modernějším řešením ve formě standardizovaného Ethernetu. Touto volbou jednak adoptuje širokou škálu fyzických vrstev od mědi až po optiku a od 100Mbit verze až po to co i teprve přijde a bude následně adoptováno standardem.

Profinet IO je zaměřen jak předznamenává jeho přídomek IO (Input Output device) na propojování primárních automatizačních zařízení (senzorů, akčních členů, PLC, HMI apod.). Tato primární automatizační zařízení jsou spojována s nejnižší vrstvou v hierarchii průmyslové automatizace a mají omezené unifikované funkce se striktní vazbou na reálný čas.

Mezi takové funkce patří obsluhování množství výstupů a vstupů, diagnostika a zabezpečení ve formě alarmů, sekundárním dopadem je pak úspora kabeláže a částečná distribuce řízení.

Existuje ještě druhá varianta označovaná Profinet CBA vycházející z Profibusu FMS, která je zaměřena na propojování automatizačních celků a v hierarchii průmyslové automatizace je tak výše než Profibus IO. Profibus CBA jak jeho přídomek (Component Based Automatisation) předjímá, slouží k propojování složitějších automatizačních celků, které jsou zde reprezentovány ve formě jeho propojovaných komponent. Tyto komponenty tak představují nadefinovaná rozhraní pro vzájemnou interakci (ovládání a vlastní stav) jednotlivých automatizačních celků.

Trendy ve vývoji

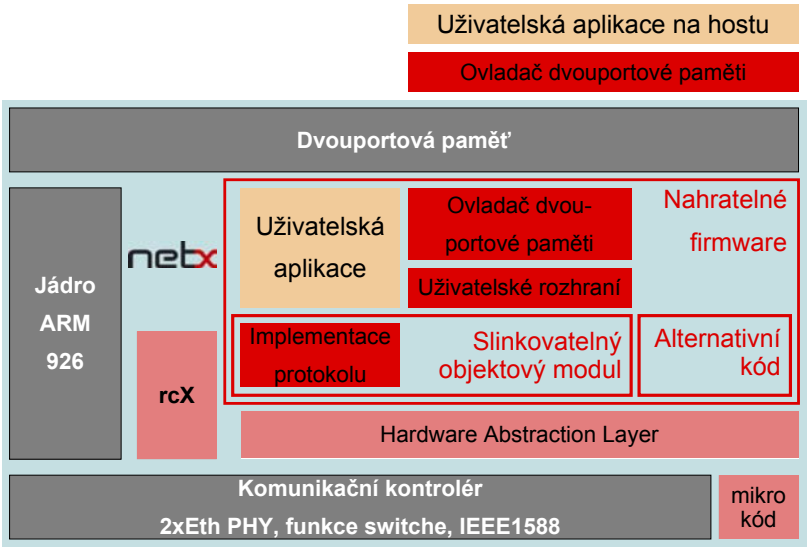
Komunikační stack

Jsme-li postaveni před úkol umožnit automatizačnímu zařízení komunikaci přes Profinet IO můžeme se tohoto úkolu shostit v zásadě třemi možnými přístupy:

- Vytvořením vlastní implementace Profinet IO komunikačního stacku. Tato možnost je opodstatnitelná pokavaď buď žádná jiná možnost neexistuje, nebo se ekonomicky nevyplatí a nebo nedokáže splnit požadované parametry.
- Implementujeme exitující Profinet IO komunikační stack. Tato možnost je vhodná u větších sérií, kde se fixní licenční náklady a náklady na vývoj rozpustí v dostatečném množství vyrobených zařízení. Jelikož jsou dodávané Profinet IO komunikační stacky dodávané ve formě zdrojových kódů v jazyce C, je zajištěna přenositelnost. Komunikační stacky pro Profinet IO dodávají firmy:

- **Siemens AG**, resp. kompetenční centrum **ComDeC.** U tohoto řešení není pro uživatelské rozhraní vytvořeno žádné abstraktní rozhraní. Služby protistrany jsou používány prostřednictvím volání jejich obslužných funkcí
- **Softing AG**. U tohoto komunikačního stacku je vytvořeno rozhraní mezi ním a uživatelskou aplikací ve formě sdílené paměti, které je navíc z uživatelského pohledu totožné i pro ostatní komunikační stacky: EtherNet/IP a Modbus/TCP.

- Použitím komunikačního modulu nebo koprocesoru. Toto je vhodná varianta u malých sérií, kde licenční poplatek a náklady na vývoj rozpočtené do malého rozsahu série stále převyšují zvýšenou cenu tohoto řešení. Nevýhodným se naopak stává, jeli vyžadovaná širší škála podporovaných komunikačních protokolů nebo při jiných zejména konstrukčních omezeních. Takovéto řešení založená na komunikačních modulech nebo koprocesorech již nabízejí např. firmy:
  - **Softing AG**. Tato firma dodává jak komunikační koprocesor, tak i na něm postavený komunikační modul. Obojí je postaveno na FPGA Altera III se dvěmi Ethernetovými porty a se syntetizovaným jádrem procesoru NIOS II, na kterém běží operační systém eCos nebo embOS. Na platformě je použit komunikační stack v objektové formě vyvinutý toutéž firmou, a ke kterému je přistupováno aplikací přes sdílenou paměť nebo pomocí sériového rozhranní skrze implementované rozhraní.
  - **Hilscher Gmbh.** Tato firma moduly i samotné procesory postavené na jádře ARM 926 obohaceném o dvě Ethernetové rozhraní rozšířené o nezbytné funkce vyžadovanými podporovanými komunikačními protokoly (IEEE 1588, funkce switche, funkce hubu pro EtherCat). Procesor rovněž obsahuje čtyři rozhraní CAN a další sériová rozhraní pro možnost připojení ostatních rozšířených protokolů. V současné době je s tímto řešení možné připojit uživatelské zařízení k průmyslovým sítím pomocí dodávaných stacků ASinterface, CANopen, CC-link, DeviceNet, EtherCat, Ethernet/IP, Ethernet Powerlink, Interbus, Profinet a Profibus, Sercos III.



Obr. 1, Možnosti využití komunikačního procesoru fy. Hilscher

V závislosti na licenčních nákladech může být tento procesor provozován ve funkci komunikačního koprocesoru s nahraným pevně daným firmware vybraného implementovaného komunikačního protokolu. V této verzi,

zatížené pouze poplatkem za dodaný kus, probíhá komunikace s hostitelským procesorem uživatelské aplikace pomocí dvoubranové paměti RAM. Tato komunikace je zde rozlišena na dva druhy. První typ pro acyklickou komunikaci je realizovaný frontami zpráv. Druhý typ zaměřený na cyklické vyměňování obrazů vstupů a výstupů je realizován samostatnými vyhrazenými oblastmi paměti. Hostitelský procesor může při čekání na příchozí zprávu využít buď dotazování nebo implementovanou synchronizační entitu pomocí vyhrazené části paměti.
Ostatní řešení nabízená firmou Hilscher nabízejí kromě prodeje holého procesoru i další varianty jako je umístění uživatelské aplikace přímo do procesoru slinkováním s modulem implementovaného protokolu či s jiným alternativním kódem, nebo použití např. Linuxu nebo WinCE za pomoci dodávaného BSP namísto standardně dodávaného realtime operačního systému rcX firmy Hilscher. Od takovýchto řešení se ovšem odvíjí i licenční politika, která u těchto variant zahrnuje nenulové počáteční poplatky a je tak mimo dosah rentability kusových sérií připojovaných zařízení.

Operační systémy

Druhým řešeným dilematem při návrhu zařízení je volba operačního systému, jsou opět v zásadě čtyři formy přístupu:

- Nepoužit žádný OS. Tato varianta dnes prakticky připadá v úvahu pouze u vestavěných zařízení realizovaných na mikrokontrolérech a i v tomto segmentu je tato možnost stále vzácnější, neboť se stále objevují i pro nejmenší kontroléry buď plnohodnotné minimalistické OS nebo alespoň pseudo OS, které tak programátorům nabízí alespoň základní podporu při tvorbě aplikací obvyklým unifikovaným způsobem.
- Použití specifického OS. Do této kategorie spadá i převážná většina OS pro vestavěná zařízení. Za charakteristický znak lze považovat jistou stálost konfigurace, kdy s aplikací tvoří pevný a neměnný celek, tvoří totiž její podporu a odstínění od implementačně závislých záležitostí. Jelikož tento OS podporuje pouze jedinou statickou předem danou aplikaci (vykonávaný proces) pozbývá významu implementace podpory virtuální paměti. Díky tomuto a dalším zjednodušením na základě zmíněné statické vazby aplikace jsou tyto OS relativně jednoduché a tím pádem i s menšími nároky na zdroje, druhým důsledkem relativní jednoduchosti a statičnosti je také jednodušší zajistitelnost deterministického chování celého takového systému. Právě z lépe dosažitelné determinističnosti je většina OS realizovaná na tomto modelu a navíc právě z důvodu minimálních systémových nároků je tato koncepce nejrozšířenější u vestavěných zařízení.
- Použití všeobecného OS. Tyto OS obecně nabízejí nejširší podporu programátora. Mezi základní podporu u takovýchto OS patří i podpora systému virtuální paměti a tím i dynamičnosti jeho užívání např. spouštěním a ukončováním běhu uživatelských programů (současného běhu více procesů) s přirozenou možností jejich separátní výměny. U takovýchto systémů jsou přirozeně díky jejich složitosti významně vyšší systémové nároky a nejen kvůli možnosti negativního projevu uživatelské aplikace i těžce splnitelné požadavky determinismu.
- Použití „zlaté střední“ cesty. V případě, že jsou v jediném systému vyžadovány přednosti plnohodnotného OS pro část úloh, ale zároveň je i potřeba současně deterministicky vykonávat zbývající část úloh, přistupuje se dnes často k modelu

specifického OS, který zajišťuje determinismus pro realtime část aplikace a zároveň je jednou z jeho úloh i virtualizovaný plnohodnotný OS, ve kterém jsou vykonávány, a kterým poskytuje i svoji plnou a širokou podporu, zbylé aplikace.

Analýza problému a definice úloh

Má práce se zabývá vývojem zařízení na bázi dodaného komunikačního stacku od firmy Siemens a na základě volby dvou typů operačních systémů je tak předem předurčena pro vývoj zařízení pro dvě specifické oblasti použití.

- ThreadX je typem specifického OS se zaměřením na realtime aplikace vestavěných zařízení. Podporuje pouze staticky vázanou aplikaci se kterou sdílí společný adresní prostor. Na základě tohoto OS jsou tak především realizovatelné časově stálé aplikace jako jsou distribuovaná procesní zařízení se kterými se běžně setkáváme.
- Linux je representantem plnohodnotného OS původně zacíleného na nasazení v osobních počítačích odkud si přinesl podporu více současně spuštěných procesů pro více uživatelů. A ač se snaží Linux prosadit i do oblasti vestavěných systémů snížením systémových nároků a zlepšením odezvy zavedením preempce při vykonávání kódu jádra, zatím stále plně nesplňuje požadavky na realtime OS. Na základě tohoto OS se tak především realizují různá zařízení, která především interagují s uživatelem jako jsou HMI a různé jiné supervizorská zařízení a testery.

Přenášení aplikací

Přenášením aplikace (portování) je všeobecně míněno její převádění původního do nového cílového prostředí. V dnešní době téměř výlučného používání vyšších jazyků, mezi které patří i jazyk C, odpadá dřívější nezbytné kompletní přepisování celých programů. Ovšem ani při využití vyšších jazyků se neumíme úplně vyhnout nutnosti změň v kódu. Tyto změny kódu můžou být vyvolány v zásadě dvěmi příčinami:

- Změna v rozhraních, jenž program využívá – knihovní funkce. Tato situace nemusí nastat ani při přenášení programu na jinačí systém, ale postačí aktualizace nějaké komponenty tohoto systému, která tak může změnit rozhraní vůči našemu programu a my musíme tuto změnu reflektovat. V takovém případě programátor provádějící úpravy ocení vytvořenou abstraktní vrstvu aplikace, jejíž hlavní funkce spočívá v odstínění zbytku programu od těchto implementačně závislých funkcí. Takovéto abstraktní rozraní je vhodné pro snadnou údržbu a následné úpravy sdružit do jednoho místa. Při změně prostředí tak postačí změna na jediném místě. Druhou výhodou centralizace využívaných rozhraní lze využít k evidenci používaných zdrojů. Využívané zdroje tak mohou být evidovány a zpravidla pomocí svého identifikačního čísla referovány. Tímto přístupem je jednak zjednodušeno a unifikováno používání těchto zdrojů, a za druhé, je díky evidenci umožněn snadný a bezpečný úklid těchto prostředků po skončení aplikace.
- Přechodem na rozdílný programovací model. U tohoto případu můžou vyvolané změny vyústit v zásadnější změnu v architektuře celé aplikace.

Implementace

Při integrování komunikačního stacku do operačního systému musí být obecně vzaty do úvahy jeho specifické nároky. U realtime komunikačního protokolu jsou to především

nároky nízké zpoždění mezi obdržením rámce a jeho zpracováním a naopak. Za tímto účelem je tak nutno obecně provést tři základní opatření:

- Obejít TCP/IP stack systému, který jinak představuje nepřijatelné zpoždění. Nejrozšířenějším takovýmto možným opatřením je zaregistrování nové obslužné funkce pro filtrování Profinet IO rámců přímo po jejich obdržení z ovladače síťového zařízení a zasílání odchozích rámců nazpět přímo přes tento ovladač.
- Pro zvýšení odezvy na zpracování je, díky způsobu realizace stacku pomocí návrhu implementace modelu událostmi řízené aplikace, nutno jednak vhodně nastavit prioritu vláken stacku a minimalizovat latenci plánovače jádra optimální volbou frekvence systémového časovače a u Linuxu zapnutím preempce v jádře.

ThreadX

U vývoje zařízení pro tento OS, pro něž je připravena vzorová implementace, se má práce omezila pouze na úpravu komunikačního stacku vyvolanou použitím jiného překladače a jiné verze OS ve které je jinak řešeno obcházení TCP/IP stacku. Z hlediska přenášení aplikace se tak jedná pouze úpravy použitých rozhraní. Toto je věnována samotná kapitola.

Linux

S použitím komunikačního stacku pod Linuxem souvisí dilema, kam jej umístit, neboť Linux podporuje virtualizaci paměti a tak jak jádro OS, tak každý zpuštěný proces běží ve svém vlastním virtuálním prostoru. Tímto tak předně dochází k hlavnímu dělení na prostor jádra a na uživatelský prostor vykonávaných aplikací. Jsou tak v podstatě tři možnosti umístění komunikačního stacku, a to do:

- Uživatelského prostoru. S touto variantou souvisí výhody v oblasti lepší podpory programátora vývojovými nástroji i dostupnými knihovnami a tím pádem i snadnějšího a rychlejšího vývoje aplikace. Z hlediska bezpečnosti je díky běhu aplikace je ve svém virtuálním prostoru je tak zajištěna stabilita systému.
- Prostoru jádra. V této variantě ztrácí programátor některé výhody vývoje z uživatelského prostoru, jako je podpora knihovních funkcí a horší podpora ladění, což je naopak vyváženo rychlejší odezvou zde umístěné aplikace díky přímé dostupnosti požadovaných rozhraní a eliminací přepínání kontextů při zpracovávání reakce na akci přímo v rámci jádra, což je z hlediska zpoždění u virtualizované paměti obzvláště citelné. Z hlediska stability systému jsou však na vývoj v jádře kladené vysoké nároky, neboť jakákoliv chyba v tomto prostoru může ohrozit stabilitu celého systému a jakýkoliv aplikací alokovaný prostředek se ukončením jejího běhu v jádře automaticky neuvolňuje a do dalšího restartu tak zůstává nedosažitelným.
- Jak uživatelského prostoru, tak i do prostoru jádra. Tato „zlatá“ střední cesta spojuje výhody obou výše uvedených přístupů a eliminuje jejich nevýhody. Cenou za toto je poté nutnost vytvořit rozhraní mezi těmito různými prostředími. Celá aplikace je tak rozdělena na část jádra komunikačního stacku spolu s přiléhajícími komunikačními rozhraními a je umístěna do jádra, čímž je zajištěna maximální dosažitelná rychlost odezvy a jelikož až rozhraní, které není rozsáhlé, je jádro komunikačního stacku dodávané odladěné odpadají bezpečnostní a vývojářské problémy. Uživatelská část aplikace je umístěna do uživatelského prostoru, kde požívá programátor větší podpory a systém je zde na chyby v aplikaci inertní. Rychlost odezvy zařízení tímto není dotčena, neboť je dána odezvou části umístěné

v jádře, která pracuje s obrazem dat zařízení, která jsou plněna částí aplikace umístěné v uživatelském prostoru, která tak neovlivňuje přímo odezvu a k jejímu správnému fungování postačí vhodně nastavená priorita, aby mohla vždy včas aktualizovat obraz dat v jádře s daty ve skutečném zařízení.

Pro vybranou implementaci byla zvolena posledně uvedená střední cesta, protože se jedná o vývoj typu obyčejného IO zařízení, v případě vývoje zařízení isochroně komunikujícího zařízení ovšem tato verze nepřichází v úvahu, neboť i aplikace musí v tomto případě být dostupná s minimálním zpožděním a tak by vše muselo být umístěno do prostoru jádra a i to by zřejmě u Linuxu jako nerealtime systému nestačilo k úspěšné realizaci.

Z hlediska přenášení aplikace u tohoto přístupu dochází, jak k úpravám v důsledku změn v používaných rozhraních aplikace, tak z důvodu její dělby do uživatelského prostoru a prostoru jádra a tím vyvolané potřeby vytvoření propojovacího rozhraní ke změně architektury celé aplikace. Vývoji zařízení pro Linux na základě této stření cesty je věnována samotná kapitola.

ISO-OSI Referenční model

Historie standardů ISO-OSI

Ačkoliv historie počítačových sítí je datována již od počátku 60. let 20. století, tak za významný milník, z dnešního úhlu pohledu, lze označit spuštění ARPANETU v roce 1969. Tato klíčová událost byla předzvěstí následujícího bouřlivého rozvoje v tomto segmentu komunikační techniky.

Bohužel tento bouřlivý vývoj přinesl s sebou i negativa spočívající ve tvorbě nestandardizovaných, proprietárních a navzájem nekompatibilních komunikačních architektur a protokolů vyvíjených jednotlivými výrobci komunikačních zařízení (např. SNA [system network architecture] od IBM z roku 1974). Na základě tohoto vývoje vyvstala akutní potřeba vytvořit jednotnou standardizovanou sadu komunikačních protokolů a umožnit tak propojení rozdílných heterogenních systémů od různých výrobců. Tohoto úkolu se zhostila ISO, jenž za tímto účelem v roce 1977 vytvořila pracovní skupinu pro vytvoření standardů pro „propojení otevřených systémů“ (Open Systems Interconnections [OSI]). Základní prioritou této skupiny bylo vytvoření samotného rámce pro definici potřebných standardů, a tak vznikl Referenční model a související terminologie. Na základě tohoto Referenčního modelu, byla sice vyvinuta řada protokolů, ale díky své komplexnosti a složitosti nedošly významnějšího rozšíření. Tomuto vývoji také napomohlo rychlé rozšíření rodiny Internetových protokolů (IP, TCP, UDP, ARP, atd.) zejména z důvodu jejich větší jednoduchosti, související snadnější implementovatelnosti a lepší výkonnosti. Vývoj rodiny Internetových protokolů započal taktéž v roce 1977.

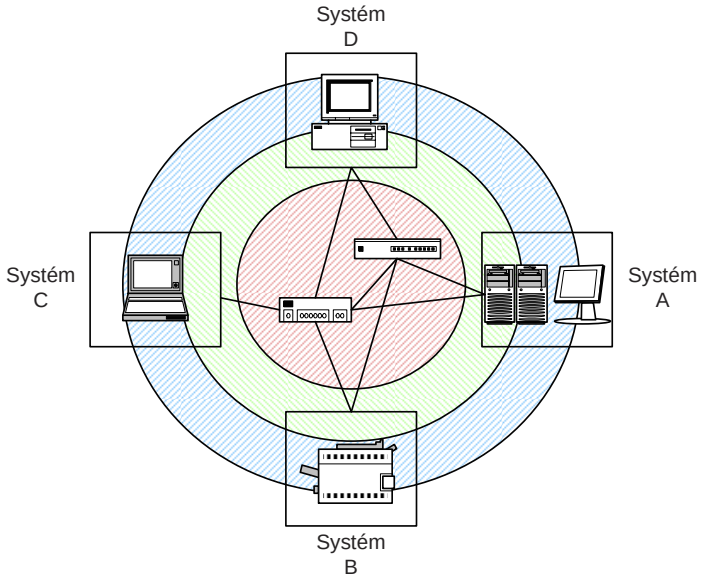
V dnešní době se ze všech standardů vyvinutých pro OSI setkáme prakticky pouze s těmi, které definují samotný Referenční model, jenž je často obecně zmiňován pouze v kontextu jím stanovených 7mi vrstev komunikačního subsystému, ačkoliv je ve skutečnosti díky své komplexnosti a obecnosti využíván i při definování nových komunikačních standardů na jím zavedeném modelu komunikace a použité terminologii.

Obecný model

Základním cílem při tvorbě OSI standardů je definovat pravidla interakce mezi propojenými systémy takovým způsobem, aby pouze vnější chování muselo odpovídat OSI architektuře, kdežto vnitřní struktura a chování propojovaného systému, ostatními připojenými systémy neviditelná, tak zůstala mimo dosah působnosti standardizace OSI. A právě na základě potřeby umožnění plného popsání vnějšího chování systému bez nutnosti vazby na jeho vnitřní implementaci vyvstala potřeba zavedení Referenčního modelu. A to takového modelu, na němž bude možno vytvořit specifikaci abstraktního systému s daným vnějším chováním a zároveň prostého jakýchkoliv implementačních, či jiných omezení. Takovým abstraktním systémem je tzv. otevřený systém, což je vyjádření v Referenčním modelu těch částí reálného systému, které jsou zapojeny do komunikace a spadají tak do působnosti OSI. Reálným otevřeným systémem je pak takový reálný systém, který splňuje dané požadavky OSI.

Klíčovým přístupem v Referenčnímu modelu je použití vrstev, které umožňují nahlížet na celou síť propojených otevřených systémů, jako na jeden propojený systém složený ze slupek – vrstev realizujících nějakou obdobnou funkci, kdy každá tato jedna vrstva odděluje vyšší vrstvu od nižší, přičemž tu nižší zabaluje do sebe jejím specifickým využíváním a tak od vyšší, jenž sousedí jen s ní, izoluje. Vždy ale tato vrstva representuje

vykonávání stejných funkcí přes celý systém, jedno v kterém propojeném systému jsou vykonávány.

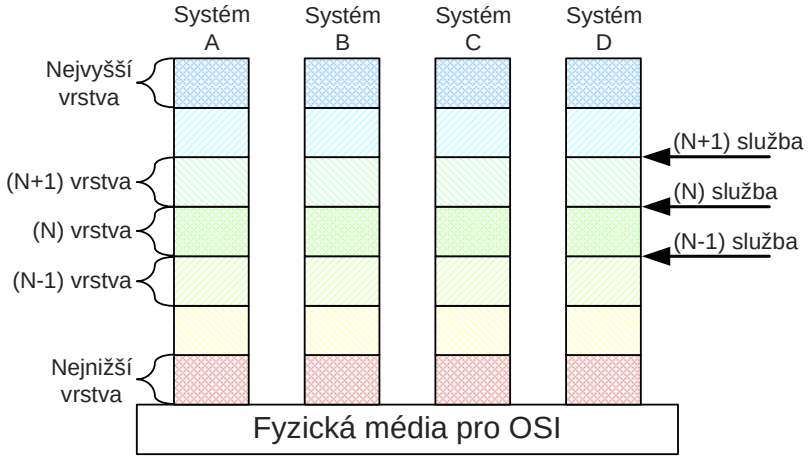


Obr. 2, Pohled na reálný systém skrze vrstvy

Alternativní pohled na takovýto slupkový model je při uspořádání jednotlivých vrstev nad sebe, kdy tyto jednotlivé vrstvy zde representují subsystémy, každý vždy jen pro jeden systém. V takovémto případě spodní subsystém sousedí s fyzickým přenosovým médiem a vrchní naopak s distribuovanou aplikací.

Subsystém tedy představuje v tomto modelu průsečík vrstvy a otevřeného systému a je vždy jen a pouze jeden na systém a vrstvu. Takovýto subsystém je tvořen jednou, či více, jeho entitami, kdy pak každá tato entita může příslušet pouze jednomu systému.

Entitami jsou zde chápány části programů nebo zařízení, které se účastní komunikace. Skupina entit nalézající se ve stejné vrstvě je označována jako skupina rovných entit.

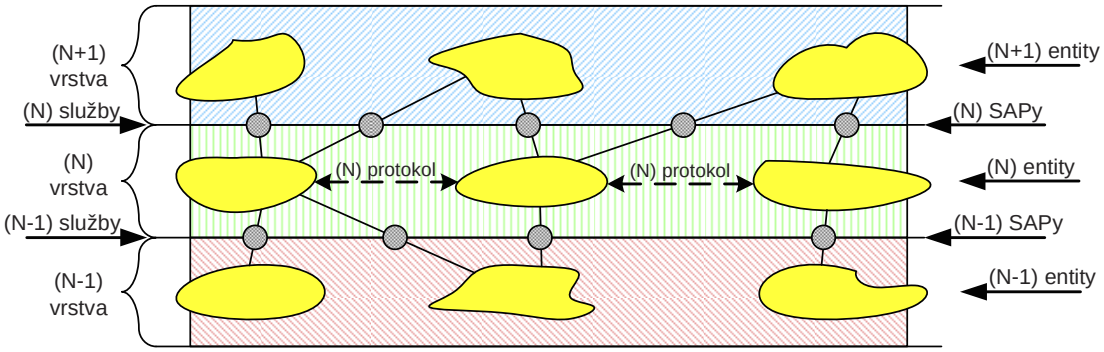


Obr. 3, Systémy, vrstvy a služby otevřeného systému

Pro orientaci mezi vrstvami je zavedena notace (N) vrstvy, která představuje jakoukoliv vrstvu. V takovéto notaci pak této vrstvě nadřazená vrstva je označována jako (N+1) vrstva a vrstva podřízená, jako (N-1) vrstva. Tato notace se vztahuje i na ostatní terminologii a tak je (N) vrstva složena z (N) subsystémů a ty zase z (N) entit.

Primárním důvodem k zavádění vrstev je představa o logické i funkční dekompozici systému do menších lépe a snázeji definovatelných a spravovatelných částí, zde lokálně představovanými subsystémy a globálně celými vrstvami. Tento přístup umožňuje strukturovaný přístup k řešení komunikace, kdy jednotlivé komunikační funkce jsou přiřčeny jednotlivým vrstvám a ty jsou je pak schopny izolovaně provádět, což má za důsledek možnost standardizovat rozhraní mezi takovýmito vrstvami, neboť všechny vrstvově specifické funkce jsou zapouzdřeny uvnitř vrstvy, která, viděno zvenčí, pouze přidává hodnotu nižší vrstvě a tu pak následně dává k dispozici vrstvě vyšší.

Zmíněná nabídka svých schopností vyšší vrstvě je označována jako služba, kterou poskytují entity všech vrstev, kromě té nejvyšší, která již vyšší vrstvu nemá a těžiště její činnosti spočívá v integraci distribuované aplikace s komunikačním systémem. Při tomto pohledu na entity se nerozlišuje, k jakému systému patří. Každá takováto (N) entita nabízí svou (N) službu své nadřazené (N+1) vrstvě. Prakticky (N) entita využije (N-1) služby (N-1) vrstvy, kterou se svojí přidanou hodnotou nabízí jako (N) službu (N+1) vrstvě.



Obr. 4, Uspořádání entit, SAPů a protokolů v RM

Spolupráce mezi (N+1) entitami se děje pomocí (N+1) asociací, které si mezi sebou naváží pomocí (N) služeb (N) vrstvy ve které proběhne potřebná komunikace vyžadovaná asociovanými (N+1) entitami. Komunikace entit z nejnižších vrstev probíhá prostřednictvím fyzického média, které v při tomto pohledu představuje virtuální (0) vrstvu.

V případě komunikace dvou entit stejné vrstvy umístěných v témže systému spadá tato komunikace do Lokálního prostředí systému a tato komunikace tak není zahrnuta v Architektuře OSI, čímž se stává volně upravitelnou podle potřeb implementace.

Spolupráce při provádění (N) funkcí, což jsou aktivity (N) entit, v (N) vrstvě mezi (N) entitami a za použití (N-1) služeb je specifikovaná (N) protokolem. Tyto (N) funkce tvoří přidanou hodnotu, která se následně dává k dispozici (N+1) vrstvě skrze (N) služby.

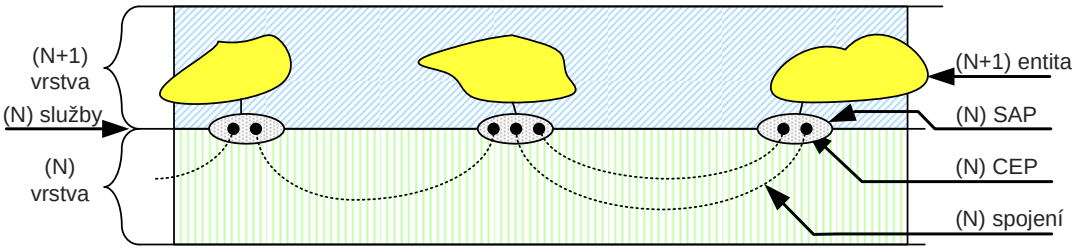
Služba je vždy nabízena na rozraní vrstev, tj. (N) služba je nabízena (N+1) vrstvě na jejím rozhranní s (N) vrstvou a tento bod logického rozhranní je označován jako (N) Přístupový bod ke službě ((N)SAP – Service Access Point).

Platí zde zásada, že každý (N) SAP může být obsluhován pouze jednou (N) entitou, která však může zprostředkovávat více (N) SAPů, naopak (N+1) entita může používat více (N) SAPů a to dokonce i těch obsluhovaných jednou (N) entitou aniž by tato o této skutečnosti věděla.

Základní službou, vyskytující se v každé vrstvě musí být možnost vytvoření asociace mezi dvěmi či více (N) SAPy, čímž se umožní přenos dat mezi dvěmi, či více (N+1) službami.

Na základě této kardinality vztahu mezi SAPy rozlišujeme propojení typu point-to-point v případě kardinality 1:1 nebo typu multiendpoint v případě kardinality 1:N, což odpovídá např. broadcastu.

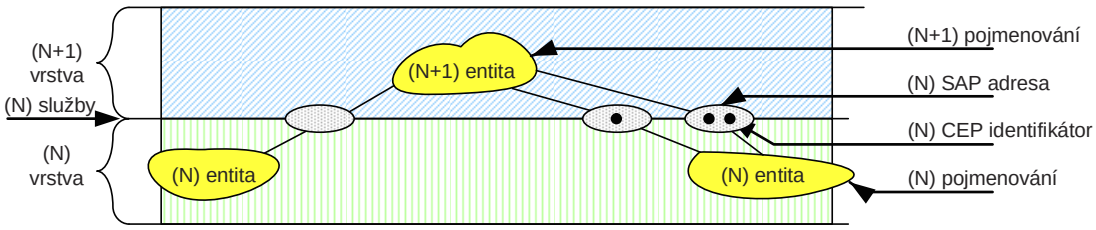
Při vzniku propojení mezi dvěmi, či více SAPy se vytváří v každém v něm tzv. komunikační koncový bod (CEP connection endpoin), takto propojený SAP může obsahovat více CEPů, kdy každý identifikuje individuální propojení zakončené v tomto SAPu.



Obr. 5, Spojení a koncové body spojení (CEPy)

Při takto zvoleném uspořádání vyvstává požadavek na schopnost identifikovat žádaný objekt uvnitř otevřeného systému pro účely adresace. Jednoznačně identifikovatelné musí být tedy jak objekty vyskytující se na rozhranní vrstev (SAPy, CEP), tak i objekty ve vrstvách zapouzdřené (entity).

Identifikace jednotlivých entit probíhá pomocí jim přiděleného jednoznačného globálního pojmenování. SAPy jsou identifikovatelné svojí unikátní adresou a CEPy jsou jednoznačně identifikovatelné adresou SAPu v němž se nachází a CEP identifikátorem, jenž v obou, či více, komunikujících entitách v nadřazených vrstvách identifikuje toto propojení.



Obr. 6, Pojmenování, adresy a CEP identifikátor

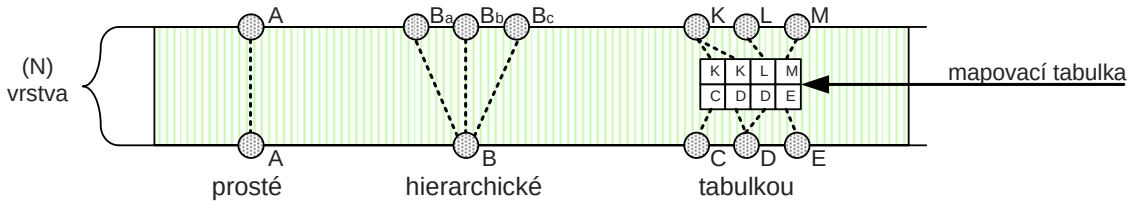
V návaznosti na takto zavedenou identifikaci vyvstala potřeba mapování požadavků entit, které jsou identifikovány pomocí svého unikátního pojmenování, pomocí služeb nižší vrstvy, tj. pomocí SAPů, které jsou identifikovány adresou. Toto mapování adres SAPů na názvy entit je obecně prováděno skrze daný adresář.

(N) pojmenování	(N-1) adresa
A	123
B	234
C	345
D	456
E	567
…	…

Obr. 7, Příklad (N) adresáře

Nastane-li situace, že (N+1) entita potřebuje navázat asociaci s jinou (N+1) entitou provede mapování identifikátoru vzdálené (N+1) entity na adresu (N) SAPu, jenž po obdržení adresy vzdáleného (N) SAPu příslušejícího vzdálené (N+1) entitě vytvoří spojení, které lze nadále identifikovat pomocí vytvořených (N) CEPů.

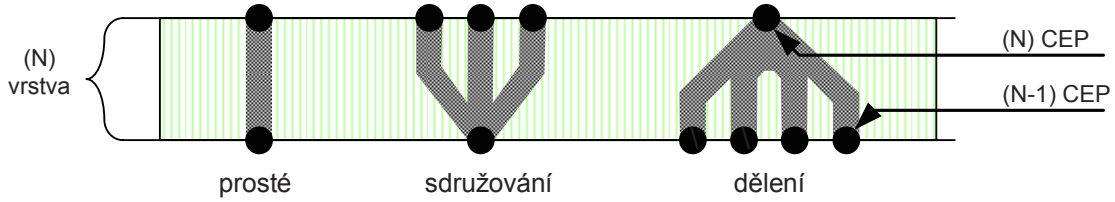
V případě, že (N) entita zprostředkovávající (N) službu přes (N) SAP vyvolaný (N+1) entitou v úmyslu navázání asociace se vzdálenou (N+1) entitou, nemá sama asociaci s odpovídající (N) entitou příslušející vzdálené (N+1) entitě, musí sama vyvolat požadavek vytvoření této asociace pomocí mapováním zjištěného příslušejícího (N-1) SAPu, neboť toto je podstata fungování vrstvově strukturovaného systému.



Obr. 8, Mapování mezi adresami

Vyvstane-li požadavek od (N+1) spojení na vytvoření nebo připojení ke stávajícímu (N) spojení, může nabýt korespondence (N) spojení na (N+1) spojení tří základních typů.

- Prosté spojení
- Sdružování spojení
- Dělení spojení

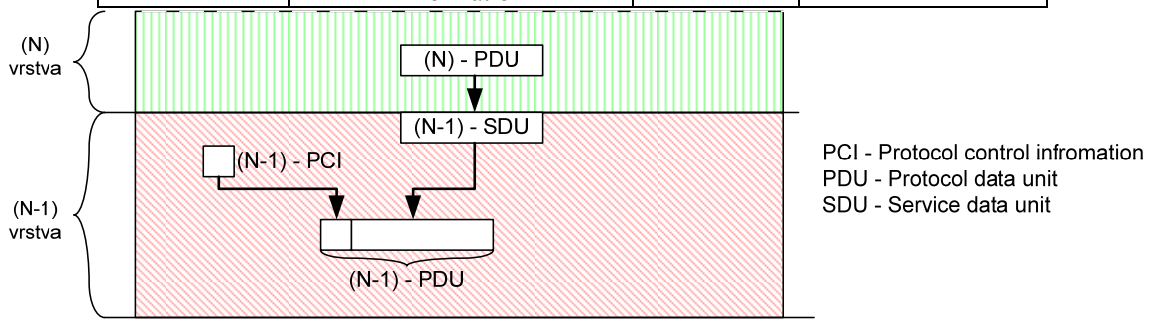


Obr. 9, Korespondence mezi spojeními

Přenos dat

Po předcházejícím popisu principů komunikace je možno nastínit i samotnou výměnu dat, která probíhá ve dvou rovinách. Virtuálně jsou data vyměňována mezi (N) rovnými entitami, což však v modelu vrstvové architektury není prakticky možné. V praxi může docházet k výměně dat pouze mezi přilehlými subsystémy v jednotlivých otevřených systémech a virtuální komunikace mezi (N) rovnými entitami v (N) asociaci je tak přes služby nižší vrstvy převedena na virtuální komunikaci mezi (N-1) rovnými entitami v (N-1) asociaci, což se opakuje přes všechny nižší vrstvy, až bude pomocí (0) vrstvy – fyzického média, reálnou komunikaci uskutečnit. Kromě umožňování virtuální komunikace (N) rovnými entitami, dochází k výměně dat mezi přilehlými vrstvami i z důvodu samotného řízení a koordinace jejich činnosti. K této vertikální komunikaci dochází na pomezí (N-1) vrstvy voláním (N-1) služby předáním příslušející (N-1) SDU neboli (N) PDU.

Rovina \ Účel	Řízení	Data	Sdružené
(N) – (N) rovné entity	(N) PCI Protocol Control Information	(N) UD User Data	(N) PDU Protocol Data Units
(N) – (N-1) přilehlé vrstvy	(N-1) ICI Interface Control Information	(N-1) ID Interface Data	(N-1) IDU Interface Data Unit



Obr. 10, Logické vztahy mezi datovými pakety v přiléhajících vrstvách

Správa

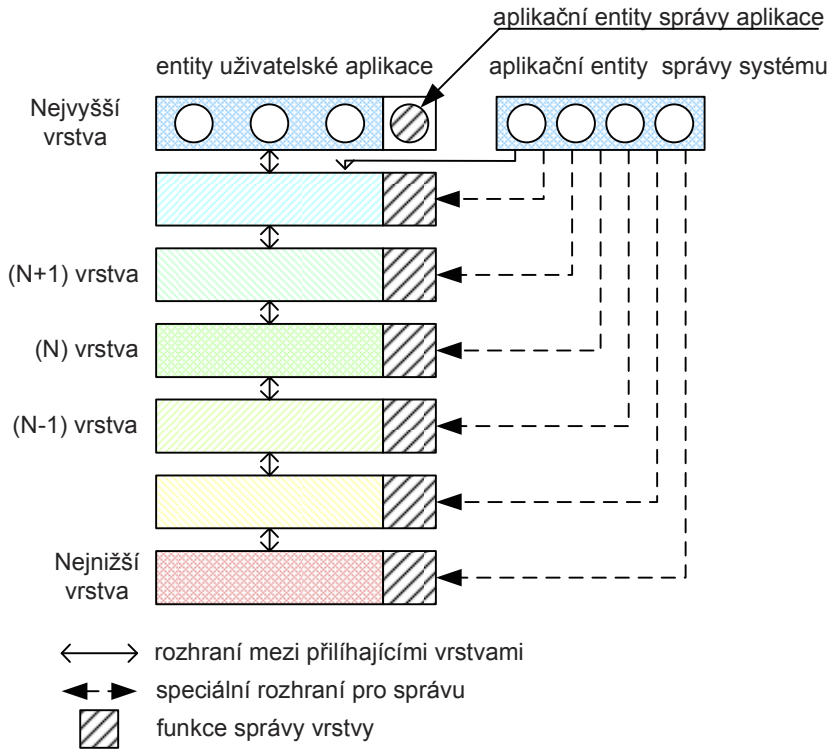
O takové aspekty fungování komunikačního systému jakými jsou jeho:

- Spouštění, zastavování a restartování,
- Konfigurace,
- Diagnostika,
- Monitoring a Statistika
- a další funkce řízení a správy,

se stará a spravuje je management komunikačního systému který je v Referenčním modelu rozdělen do dvou částí na:

- řízení aplikace
- řízení systému

Obě dvě funkcionality jsou v Referenčním modelu umístěny do Aplikační vrstvy tak, aby byly přímo dostupné Aplikačnímu procesu, což je ta část distribuované aplikace, která zasahuje do Aplikační vrstvy. Rozdíl mezi nimi je dán především rozdílnými kompetencemi, kdy funkcionalita Aplikačního řízení spravuje Aplikační vrstvu a funkcionalita Systémového řízení má na starosti zbylých 6 vrstev referenčního modelu ke kterým nemusí přistupovat kaskádně po vrstvách, ale díky tomu, že spadá do Lokálního prostředí může přistupovat ke každé vrstvě přímo.



Obr. 11, Reprezentace funkcí pro správu

Při tvorbě Referenčního modelu bylo navrženo, že otevřený systém se bude strukturalizovat do 7mi vrstev, kdy každá jedna přispívá svojí specifickou funkcí k možnosti propojení celého systému. Toto rozhodnutí bylo učiněno od zeleného stolu ve snaze vytvořit takovou sadu podobně vytížených jednotlivých vrstev, kdy každá tato representuje a zahrnuje skupinu obdobných podobných funkcí, čím se zároveň minimalizuje komunikace mezi vrstvami.

Byly nadefinovány vrstvy:

- 7. vrstva : Aplikační (Application Layer)

Aplikační vrstva je umístěna nejvýše v OSI Architektuře. Protože nemá již žádnou nadřazenou vrstvu, tak protokoly této vrstvy přímo slouží koncovému uživateli včetně nezbytného zajišťování správy komunikačního systému.

Aplikace je složena ze spolupracujících aplikačních procesů, které komunikují navzájem v souladu s protokolem aplikační vrstvy. Aplikační proces je tak z hlediska Referenčního modelu jediným skutečným zdrojem a příjemcem dat. Část aplikačního procesu se projevuje v aplikační vrstvě formou provádění aplikačního protokolu, tj. ve formě vyvolané aplikační entity. Zbytek aplikačního procesu je mimo zájem Referenčního modelu. Aplikace nebo aplikační proces může být jakéhokoliv typu. Pro potřeby Referenčního modelu postačí pouze, že vytváří a přijímá data. To jak s nimi nakládá (např. terminál s lidskou obsluhou, datový server, ovládaná technologie, atd.) a k čemu přenášená data slouží a co představují je mimo rámec zájmu OSI.

- 6. vrstva : Presentační (Presentation Layer)

Protože záměrem OSI bylo vytvořit komunikační systém pro propojení heterogenních systémů, vyvstala potřeba Presentační vrstvy přizpůsobování implementačně závislých záležitostí jakými jsou např. kódování čísel, znaků a podobně.

Účelem presentační vrstvy je provádění množiny služeb, které mohou být vybírány na základě požadavků aplikační vrstvy pro umožnění interpretování významu vyměňovaných dat. Tyto služby tak řídí přenos, zobrazování (transformaci), kontrolu strukturovaných dat.

Díky použití služeb presentační vrstvy je umožněno, aby různé aplikace v heterogenním OSI prostředí mohly spolu komunikovat bez nežádoucích omezení ve variabilitě rozhranní a nutnosti modifikace samotných aplikací.

- 5. vrstva : Relační (Session Layer)

Relační vrstva je navržena k podpoře komunikace mezi kooperujícími presentačními entitami. Za tímto účelem poskytuje Relační vrstva dvě služby:

  - Službu administrování relace, jenž je zaměřena na vytváření svázáním a rušení rozvázáním vazeb dvou presentačních entit.
  - Službu dialogu relace, která řídí výměnu dat, ohraničuje a synchronizuje operace s daty mezi dvěmi presentačními entitami.

- 4. vrstva : Transportní (Transport Layer)

K vytvoření Transportní vrstvy vedl požadavek na vytvoření univerzální transportní služby pro vyšší vrstvy za použití služeb vrstev nižších. Transportní vrstva poskytuje transparentní přenos dat mezi entitami Relační vrstvy, které tímto osvobozuje od potřeby uvažovat detaily potřebných prostředků a postupů, kterými je spolehlivá a efektivní datová výměna zajišťována. Transportní vrstva slouží taktéž k optimalizaci užívání dostupných komunikačních prostředků pro zajištění potřebného výkonu, jenž vyžaduje každé spojení mezi relačními entitami, při minimálních nákladech.

- 3. vrstva : Síťová (Network Layer)

Síťová vrstva je určena k poskytování funkčních a procedurálních prostředky k výměně síťových servisních datových paketů (NSDU) mezi dvěmi transportními entitami přes síťové spojení. Slouží k odstínění transportních entit od síťové problematiky, jenž je tvořena např. směrováním a přepínáním.

- 2. vrstva : Spojová (Data Link Layer)

Účelem Spojové vrstvy poskytovat funkční a procedurální prostředky k zřízení, udržení a uvolnění datového spoje mezi síťovými entitami.

- 1. vrstva : Fyzická (Physical Layer)

Fyzická vrstva poskytuje elektrickou, mechanickou a funkční specifikaci a postupy pro zřízení, udržení a uvolnění fyzického spojení mezi entitami datového spoje.

OSI služby

Jelikož jsou služby jedinou interakcí mezi dvěmi sousedními vrstvami představují tak ideálního kandidáta na definování tohoto rozhraní a vytvoření tak unifikovaného přístupu ke službám dané vrstvy ať realizuje jakýkoliv protokol. V obecném modelu Referenčního modelu jsou však zavedeny spíše ze strany protokolu, kdy je služba representována SDU předaného přes příslušející SAP na rozhraní vrstev.

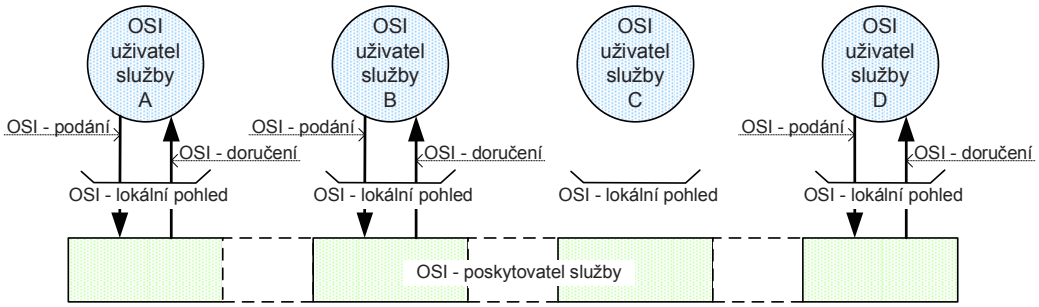
Za účelem možnosti obecné definice rozraní zavádí OSI ještě druhý pohled na služby, a to z pohledu jejich obecného uživatele a obecného poskytovatele. Uživatelem služby je v tomto případě jediná entita z vrstvy nad rozhraním v jednom jediném otevřeném systému a poskytovatel naopak zase obecně tvořen všemi komunikujícími entitami z vrstvy pod

rozhraním ve všech propojených otevřených systémech. Uživatel a poskytovatel služby tak spolu interagují v tzv. lokálním pohledu příslušejícímu uživateli služby.

Skrze rozhraní lokálního pohledu dochází k výměně servisních primitiv. Entita která servisní primitivu vysílá je nazývána žadatel a servisní entita, které je servisní primitiva určena je nazývána příjemcem. Servisní primitiva je abstraktní implementačně nezávislá reprezentace interakce mezi uživatelem a provozovatelem služby ve významu sémantiky a významů přenášených parametrů a požadované akce.

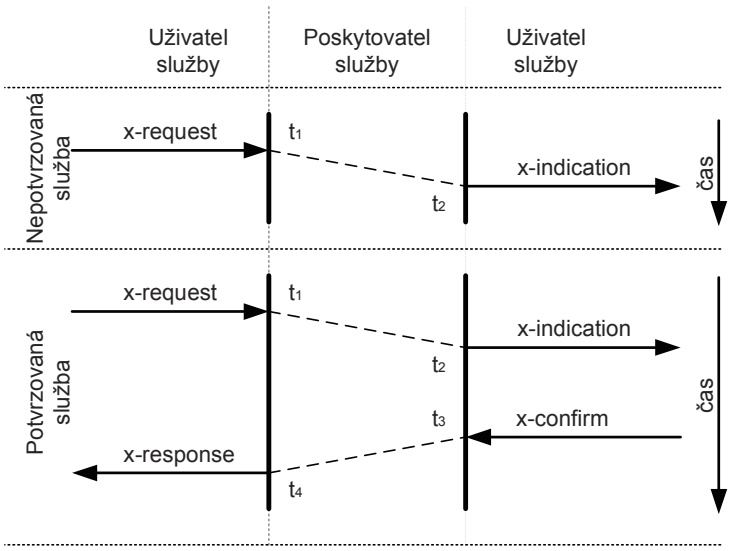
Jsou definovány čtyři typy servisních primitiv:

- žádost (request) – podaná žadatelem
- indikování (indication) – doručená příjemci
- odpověď (response) – podaná příjemcem
- potvrzení (confirm) – doručená žadateli



Obr. 12, OSI Model služeb

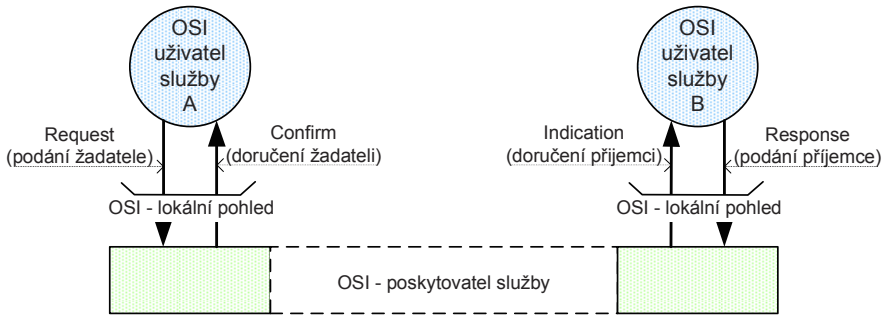
Pro vzájemnou komunikaci využívají (N) entity (N-1) služeb, které můžou být jednak potvrzované u spojovaných služeb, tak i nepotvrzované. Definice (N-1) služby udává její sémantiku jak z pohledu žádající tak i odpovídající (N) entity. Služba může být symetrická, jsou-li servisní primitivy na všech lokálních pohledech stejné, nebo asymetrická liší-li se. Případně může být služba typu více-rovných, kdy probíhá výměna mezi více jak dvěma uživateli této služby. Chování (N) entity spojené s užíváním (N-1) služeb je součástí specifikace (N) entity, nikoliv (N-1) služby.



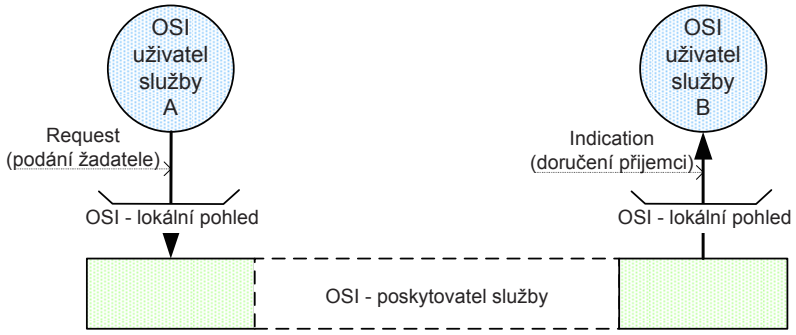
Obr. 13, Příklad časová sekvence pro nepotvrzovanou a potvrzovanou variantu

Parameter name	Req.	Ind.	Rsp.	Cnf.
Argument	M	M(=)		
AREP	M	M(=)		
API	M	M(=)		
Target AR UUID	U	U(=)		
Slot Number	M	M(=)		
Subslot Number	M	M(=)		
Index	M	M(=)		
Seq. Number	M	M(=)		
Length	M	M(=)		
Result(+)			S	S(=)
AREP			M	M(=)
Seq. Number			M	M(=)
Add Data 1			M	M(=)
Add Data 2			M	M(=)
Length			M	M(=)
Data			M	M(=)
Result(-)			S	S(=)
AREP			M	M(=)
Seq. Number			M	M(=)
Error Decode			M	M(=)
Error Code 1			M	M(=)
Error Code 2			M	M(=)
Add Data 1			M	M(=)
Add Data 2			M	M(=)

Obr. 14, Příklad definice servisní entity



Obr. 15, Příklad potvrzované služby mezi dvěmi uživateli

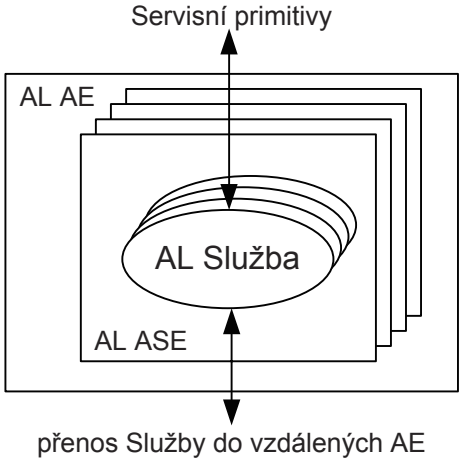


Obr. 16, Příklad nepotvrzované služby mezi dvěma uživateli

Struktura aplikační vrstvy

V prostředí otevřených systémů může být aplikace rozdělena do řady komponent a distribuována tak do řady zařízení na síti. Každá z těchto komponent je označovaná jako Aplikační proces (AP) realizovaný formou aplikační entity.

Aplikační vrstva jako nejvyšší vrstva v OSI architektuře již nezajišťuje služby ve smyslu nižších vrstev vyšší vrstvě na příslušejících SAPech. Avšak jelikož do ní spadá část komunikující aplikace, zajišťuje komunikační služby této části. Tyto komunikační služby aplikační vrstvy s podobným zaměřením jsou obecně sdružovány do větších celků nazývaných Aplikačními servisními elementy (ASE).



Obr. 17, Struktura Aplikační entity

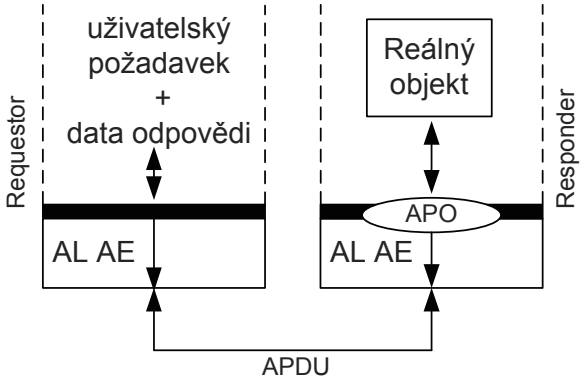
V dřívějších definicích aplikační vrstvy byly jednotlivé samostatné části aplikační entity nazývány Uživatelskými elementy, který mohly využívat nebo být dostupné pomocí běžných ASE (CASE) nebo specifických ASE (SASE), kdy CASE byly standardizovány obecně pro danou třídu aplikací a poskytovaly tak základní komunikační služby a SASE byly standardizovány pro omezený počet (často pro jedinou) aplikací. Všechny tyto ASE využívají služeb aplikaci řídícího servisního elementu (ACSE), jehož služby zajišťují správu asociací aplikace prováděné komunikací pomocí služeb Presentační služby.

Poslední platná definice struktury aplikační vrstvy zavádí objektový přístup při modelování aplikačního procesu pomocí objektů aplikační vrstvy. Obsah APů tak může být representovaný jedním nebo více objekty AP (APOy) a přistupován tak skrze jednu nebo více Aplikačních entit (AE). Pro každý AP je zde jedna a pouze jedna AE, která mu poskytuje komunikační schopnosti skrze své ASE.

APOy jsou sítovou representací specifických schopností aplikace (objektů uživatelského aplikačního procesu) AP, které jsou dostupné skrze jeho AE. Každý APO reprezentuje specifickou množinu informací a své funkčnosti AP, který je dostupný skrze služby aplikační vrstvy. APOy jsou použity k reprezentaci těchto schopností ostatním AP v systému.

Z perspektivy aplikační vrstvy je APO modelovaný jako síťově přístupný objekt obsažený uvnitř AP nebo i uvnitř jinačího APO (APO smí obsahovat APO). APOy poskytují síťovou definici pro reálné objekty obsažené v AP, které jsou tak díky nim a jejich ASE dostupné vzdáleně. Definice APO zahrnuje identifikaci služeb AL, které tak můžou být vzdálenými APy využity pro vzdálený přístup k ASO. AL služby18 jsou poskytovány komunikační entitou AP , označovanou jako AL AE. Vzdálené AP vystupující jako klient může

přistupovat ke skutečnému objektu zasláním požadavku skrze APO, který zastupuje skutečný objekt.



Obr. 18, Přístup k reálnímu objektu skrze APO

Profinet IO

Standardizace

V oblasti standardizace fieldbusů, tj. sítí pro měření a řízení v průmyslových podmínkách, je v současné době nejvýznamnějším standardem *IEC 61158 Digital data communications for measurement and control – Fieldbus for use in industrial control systéme (05/2003)*, který definuje nejvýznamnější protokoly pro procesní automatizaci. V tomto standardu jsou komunikační protokoly těchto fieldbusů brány jako modely možné komunikace, a ty jsou poté při definování dále rozloženy do sad definovaných typů použitelných služeb a protokolů. Pro orientaci v této normě byla současně vydána norma *IEC 61784 Digital data communications for measurement and control (05/2003)*, jejímž úkolem je umožnit zorientování a správný výběr příslušejících protokolů a služeb definovaných v předešlé normě. Každý definovaný fieldbusový protokol je zde jednotlivě uveden jako možný profil, jemuž je přiřazena řada jemu určených typy protokolů a služeb podle vrstev, které musí implementovat. Jsou-li využity fieldbusem i jiné, v normě nespecifikované typy služeb a protokolů, jsou tyto typy zde také uvedeny jako odkazy na jejich specifikace a norma se tak jejich popisem dále nezabývá (Ethernet, IP, UDP, RPC, atd.).

V době svého uvedení nebyla do standardu *IEC 61158* zahrnuta řada fieldbusů založených na ethernetu, které byly tak specifikovány samostatně ve formě Veřejně dostupné specifikace (PAS). Tyto fieldbusy byly zapracovány do standardu *IEC 61158* při jeho druhé revizi, která byla vydána v závěru roku 2007, současně byla vydána druhá část normy *IEC 61784-2*, která zavádí definici profilů pro nově přidané komunikační protokoly založené na spojové vrstvě Ethernetu. Jelikož k tomuto vydání došlo nedávno, nebyl jsem schopen se s těmito normami seznámit a následující popis je založen na předešlé veřejně dostupné specifikaci.

Koncept a vztah k ISO-OSI modelu

Specifikace Profinetu i všech ostatních průmyslových komunikačních sítí z *IEC 61158* je vytvořena na základě OSI ISO Referenčního modelu upraveného pro specifické potřeby standardizace těchto komunikačních protokolů v následujících směrech:

- Podstatným rozdílem oproti původnímu referenčnímu modelu je vypuštění 3. až 5. vrstvy. Tyto vrstvy buď nejsou zcela potřeba, nebo je jimi vyvíjená funkce začleněna do nejbližší sousední vrstvy, jak znázorňuje Tabulka 1.
- OSI definuje pouze jeden typ komunikačního kanálu - asociací aplikační vrstvy pro propojování APů. FAL naproti tomu definuje několik typů Aplikačních vazeb (AR), pro rozlišení a umožnění specifické komunikace aplikačním procesů (AP)
- FAL používá Spojovou vrstvu pro přenášení svých PDU, nikoliv Presentační vrstvu. Protože zde tak chybí zřejmý presentační kontext dostupný pro FAL nemůže být současně použit FAL protokol s protokoly ostatních aplikačních vrstev mezi stejným párem (skupinou) SAPů datové vrstvy.

vrstva dle OSI	vrstva a	popis dle ISO IEC 61158
7. Aplikační	Aplikační (ISO IEC 61158-5, ISO IEC 61158-6)	Zprostředkovává požadavky kladené na komunikační stack do formy uchopitelné nižšími vrstvami a naopak
6. Presenční	↑	Převádí data na/ze standardního síťového formátu
5. Relační	↑	Synchronizuje a spravuje data
4. Transportní	↑ nebo ↓	Provádí transparentní a spolehliví přenos dat
3. Síťová	↑ nebo ↓	Provádí směrování zpráv
2. Spojová	Linková (ISO IEC 61158-3, ISO IEC 61158-4)	Kontroluje přístup ke komunikačnímu médium. Provádí kontrolu chyb
1. Fyzická	Fyzická (ISO IEC 61158-2)	Enkóduje/dekóduje signály pro vysílání/příjem vhodné pro komunikační médium. Specifikuje charakteristiky komunikačního média

↑- ukazuje na vrstvu do které bude daná funkcionalita začleněna

↑ nebo ↓ - v tomto případě bude funkcionalita začleněna do nejbližší vrstvy

Obr. 19, Přechod od 7mi vrstev u OSI ke 3 u průmyslových sítí

Profinet IO

Profinet IO je v současnosti specifikován v *IEC PAS 62411 Real-time Ethernet PROFINET IO (06/2005)* a připravuje se jeho začlenění do příští revize *IEC 61158*. Již nyní už dodržuje její uspořádání. A protože je Profinet IO vystavěn na Fyzické a Spojové vrstvě Ethernetu definovaných standardy IEEE 802.1 a IEEE 802.3 omezuje se jeho specifikace pouze na Aplikační vrstvu. Součástí jeho specifikace nejsou ani služby a protokoly využívaného nespojovaného RPC, jehož specifikace vychází z technického standardu OSF.

Typy zařízení

Pro Profinet IO jsou definovány čtyři základní typy zařízení, kterými jsou IO zařízení, IO kontrolér, IO supervizor (inženýrská stanice) a parametrizační server.

Každé fyzické zařízení nadto může hostit i více než jen jedno virtuální zařízení. Tato případná virtuální zařízení nemusí být ani stejného typu a můžou tak vytvářet i jim příslušné asociace mezi sebou.

IO kontrolér

IO kontrolér je zařízení pro řízení jedním nebo více IO a vykonává:

1) cyklické funkce
   a) vyměňuje si data s asociovanými IO zařízeními
2) acyklické funkce

   a) čte diagnostické záznamy ze zařízení
   b) konfiguruje IO zařízení
   c) zapisuje parametry do IO zařízení (inicializační nebo aplikační parametry)
   d) zpracovává konfigurační a diagnostické požadavky inženýrské stanice
   e) vytváří kontext IO zařízení pomocí správy kontextu
   f) zajišťuje acyklický přístup k datovým záznamům IO zařízení
   g) zpracovává alarmů od IO zařízení
   h) zasílá alarmy IO zařízení
3) všeobecné funkce
   a) redundance
   b) dynamická rekonfigurace
   c) isochronní činnost
   d) poskytovatelko/spotřebitelská komunikace mezi IO zařízením a IO kontrolérem
   e) synchronizace času

IO supervizor

IO supervizor je inženýrské stanice pro správu, konfiguraci a sběr diagnostických data z IO kontrolérů a IO zařízení.

IO parametrizační server

IO parametrizační server je stanice serverového typu která IO zařízení jako klientovy poskytuje konfigurační data a která může také zaznamenávat jeho aplikační data a záznamy.

IO zařízení

IO zařízení jakožto kontrolérem řízené zařízení vykonává:
1) cyklické funkce
   a) vyměňuje si IO data s jedním nebo více přiřazenými IO kontroléry
   b) vyměňuje si data s asociovanými IO zařízeními
2) acyklické funkce
   a) obstarává data pro diagnostiku
   b) zpracovává konfigurační požadavky IO kontroléru
   c) poskytuje acyklický přístup k datovým záznamům pro IO kontrolér
   d) poskytuje alarmy přiřazenému IO kontroléru
   e) zpracovává parametry (inicializační nebo aplikační parametry)
   f) zpracovává konfigurační a diagnostické požadavky inženýrské stanice
   g) zpracovává alarmy od IO kontroléru
   h) zasílá alarmy IO kontroléru
3) všeobecné funkce
   a) poskytovatelko/spotřebitelská komunikace mezi IO zařízením a IO kontrolérem
   b) redundance
   c) dynamická rekonfigurace
   d) isochronní činnost
   e) synchronizace času

Reálné IO zařízení je poskládáno z různých konstrukčních částí, které při určitém použité abstrakci tvoří sadu aplikačních objektů. Kromě těchto skutečných konstrukčních částí můžou aplikační objekty odrážet i virtuální funkční části zařízení. Záměrem specifikace je tak poskytnout použitelnou sadu adresujících parametrů pro jednoznačnou a logickou adresaci těchto aplikačních objektů.

Hierarchicky je tak reálné IO zařízení z pohledu specifikace poskládáno z jedné nebo více instancí virtuálního IO zařízení, kdy pak každá tato instance obsahuje alespoň jeden nebo více aplikačních procesů, která jsou adresovány jejich identifikátorem (API). Aplikační procesy jsou strukturovány do jednoho nebo více slotů. Sloty jsou poté jemněji strukturovány na subsloty, a ty dále na kanály, které tvoří nejmenší element logické struktury IO zařízení.

Na takto nastaveném modelu vzniklo adresační schéma pro IO zařízení založené na: UUID instance zařízení, UUID rozhraní, UUID objektu, API, číslu slotu, subslotu a kanálu.

PROFINET IO komunikační model

PROFINET IO komunikační model podporuje tři typy aplikačních asociací (AR):

- Implicitní – pouze asynchronní přístup do zařízení pomocí CLRPC s možností čtení datových, diagnostických a identifikačních objektů zařízení. V tomto AR se navazují ostatní typy AR.
- IO – zpřístupňuje ostatní definované funkce pro daný typ zařízení včetně cyklické výměny dat
- Supervizorské - zpřístupňuje možnost převzetí submodulu od jiného AR, když je povoleno

Aplikační servisní elementy (ASE)

Specifikace Profinet IO obsahuje definice šablon řady ASE, které se skládají z definovaných atributů a služeb. Tyto šablony tak slouží k vytváření ASOů pro zpřístupnění jednotlivých reálných objektů IO zařízení ke komunikaci. Instanciováním ASO z šablony ASE tak vzniká v komunikačním prostředí dostupný objekt pomocí služeb daných svým typem ASE, který svými převzatými atributy popisuje reálný objekt a svojí vnitřní implementací svých metod (exportovaných služeb) k tomuto reálnému objektu přistupuje. Pro zpřístupnění celého zařízení je definována šablona kontextového ASE z něhož odvozený ASO zajišťuje vytváření a správu aplikačních asociací.

Pro profinet PROFINET IO jsou definovány následující šablony ASEů:

IO Data ASE

ASE pro IO data poskytuje sadu služeb pro přenášení cyklických IO dat. Tato data vždy náleží tomu slotu/subslotu ke kterému byla přiřazena v Kontextovém ASE. IO Data Objekty obsahují stav pro přenos. Volitelně IO Data ASE nabízí možnost sdílet vstupní Data jednoho IO zařízení s ostatními IO zařízeními. IO Data můžou být též čtena a zapisována IO supervizorem acyklicky.

Record Data ASE

ASE pro datové záznamy poskytuje sadu služeb pro přenášení data acyklicky. Aplikace IO kontroléru nebo IO supervizor si vyžádává každý přenos individuálně. ASE pro datové záznamy může být společný pro všechny APy v IO zařízení.

Logbook Data ASE

ASE datových záznamů poskytuje sadu služeb pro čtení záznamů ze zápisníku. Aplikace IO kontroléru nebo IO supervizoru si vyžadují každý přenos zvláště. ASE datových záznamů

může být společné pro všechny APy v IO zařízení

Diagnosis ASE

Diagnostické ASE poskytuje služby pro IO kontrolér nebo IO supervizor pro čtení diagnostických informací z IO zařízení.

Alarm ASE

ASE alarmů poskytuje sadu služeb pro přenášení alarmů vystavených IO zařízením nebo IO

kontrolérem. Přiřazená protistrana IO kontroléru nebo IO zařízení potvrzuje vystavený alarm.

Context ASE

Kontextové ASE poskytuje sadu služeb pro:

- přenášení parametrů a konfigurace zařízení podle popisu zařízení
- identifikuje AR koncový bod
- udržuje AR a CR parametry (časové limity, stavy, ..)
- a navazuje nebo ruší asociace navázané mezi jednotlivými APs.

Time ASE

Časové ASE poskytuje sadu služeb pro synchronizaci hodin některých nebo všech zařízení podsíti PROFINET IO.

Isochronous Mode Application ASE

ASE pro isochronně fungujících aplikace poskytuje sadu služeb pro parametrizaci a synchronizaci isochronních aplikačních procesů.

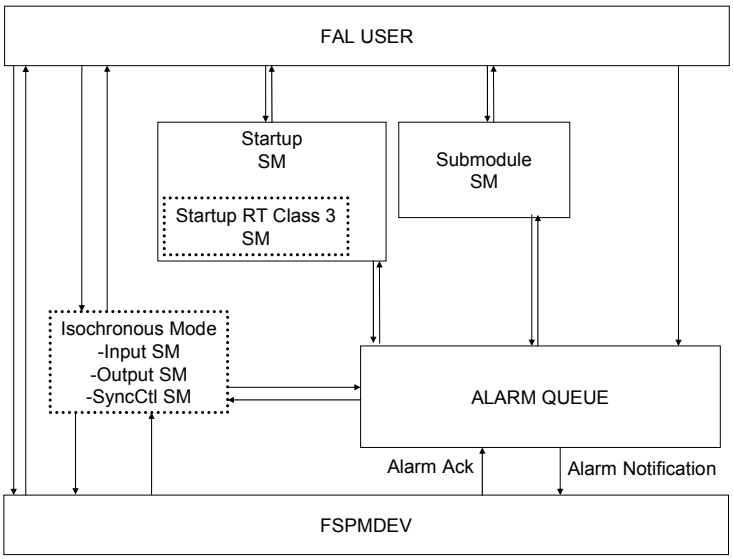
Application Relationship ASE

ASE pro aplikační (AR ASE) poskytuje popis modelu pro jednotlivé typy AR. Toto zahrnuje jejich přenosové charakteristiky stejně jako jejich současnou komunikační stav.

Physical Device Management ASE

ASE správy fyzického zařízení poskytuje sadu služeb pro správu rozběhu fyzického zařízení. To zahrnuje přiřazení jména stanice a dalších parametrů jako IP konfigurace. Toto ASE se vyskytuje v reálném zařízení pouze jednou a nezáleží tak na počtu virtuálních instancí Profinet IO zařízení v zařízení skutečně umístěných.

Stavové automaty protokolů ASE



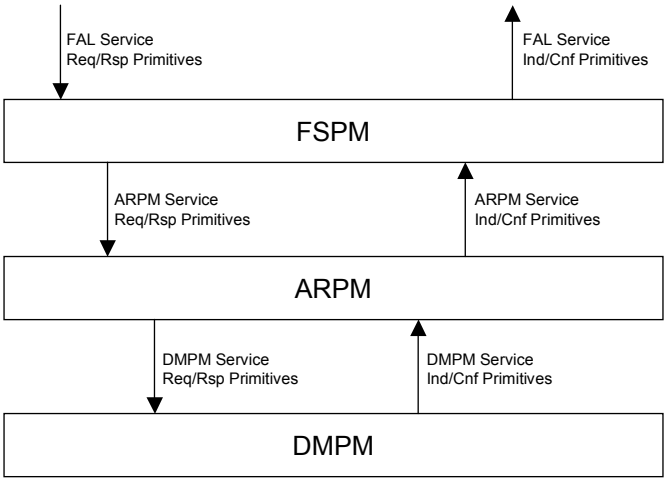
Obr. 20, Přehled struktury ASE stavových automatů

Každé zařízení musí mít implementovány následující ASE stavové automaty:

- pro spouštění – popisuje chování implementace IO zařízení během spouštění uživatelské aplikace
- pro submodulů – popisuje chování implementace IO zařízení během manipulace s moduly/submoduly
- pro frontu alarmů – popisuje chování implementace IO zařízení při vystavování nebo potvrzování alarmů

Stavové automaty protokolů FAL

Chování FAL je specifikováno pomocí třech hlavních stavových automatů: FSPM, ARPM a DMPM, jejichž architektura je rozdílná jak pro IO zařízení, tak i IO kontrolér. IO supervizor vychází z definice IO kontroléru s tím, že nemusí implementovat část jeho stavových automatů zajištujících cyklickou výměnu dat, nebude-li použit k řízení.



Obr. 21, Přehled struktury FAL stavových automatů

Fieldbus Service Protocol Machines (FSPM)

FSPM u IO zařízení nazývaná FSPMDEV je v zásadě mapovací stavový automat a koordinuje jemu podléhající stavové automaty použité pro zpracovávání různých služeb a relací aplikací. Jeho hlavní úlohou je předání služby do stavového automatu zodpovědného za tuto službu a předání potvrzení a odpovědí uživateli. Navíc je v tomto automatu použito schéma základního redundantního řízení, takže je umožněna spolupráce dvou AR.

Pro možnost tohoto rozšíření vyžaduje FSPM následující podporu:

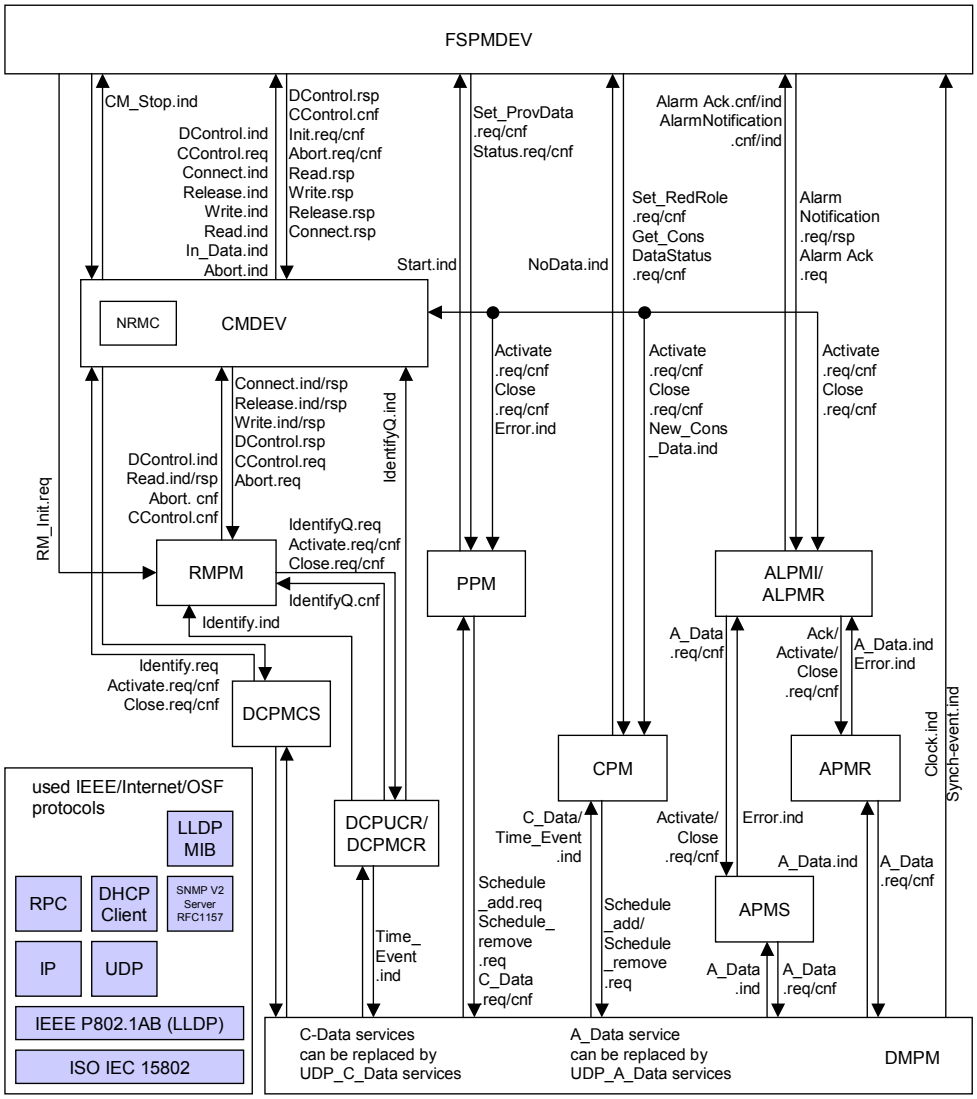
- Redundantní komunikační rozhraní do FSPM
- Koordinace primárního/záložního kanálu pomocí APDU_Status
- Přepnutí v případě jeho vyvolání
- Spolupráce datových služeb u redundantního AR

FSPM je zodpovědné za následující úkony

- Přijímání služebních primitiv od FAL uživatele služeb a jejich konverzi do FAL interních primitiv.
- Výběr ARPM stavového automatu na základě předem určeného adresování a zaslání FAL interních primitiv se servisními parametry do ARPM.
- Přijímání interních primitiv od ARPM a jejich konverzi do služebních primitiv pro uživatele FAL služeb.
- Doručování FAL servisních primitiv uživateli FAL.

Application Relationship Protocol Machinas

ARPM specifikuje sadu stavových automatů pro navázání a udržení aplikačního vztahu:



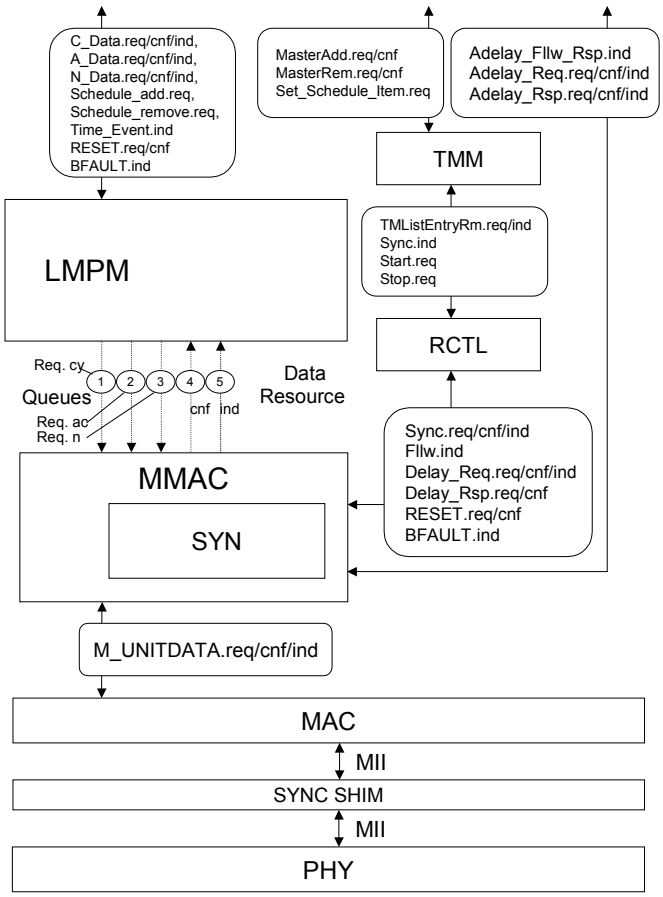
Obr. 22, Přehled struktury AR stavových automatů

Pro IO zařízení jsou v ARPM definovány následující stavové automaty:

- CMDEV správy kontextu IO zařízení (jeden na AR)
- RMPM správy zdrojů (jeden na zařízení)
- NRMC multicast rozpoznání názvu (jeden na příj. multicast zpráv)
- DCPUCR příjmu (Unicast) DCP protokolu (jeden)
- DCPMCR příjmu (Multicast) DCP protokolu (jeden)
- DHCPCL klienta DHCP protokolu (jeden)
- PPM cyklického protokolu poskytovatele (jeden na IOCR)
- CPM cyklického protokolu příjemce (jeden na IOCR)
- ALPMI protokolu alarmů (jeden nebo dva na AR)
- APMS acyklického protokolu odesílatele (jeden nebo dva na AR)
- APMR acyklického protokolu příjemce (jeden nebo dva na AR)

DLL Mapping Protocol Machines (DMPM)

DMPM zajišťuje především mapování požadavků stavových automatů ARPM na služby Spojové vrstvy za použití prioritních front a časově řízenému přístupu k médiu (odeslání rámce v definovaném intervalu). Kromě tohoto zajišťuje i služby časové synchronizace a přeposílání rámců u zařízení s více porty Ethernetu



Obr. 23, Přehled DM stavových automatů

DMPM se sestává ze stavových automatů:

- LMPM mapování na Linkovou vrstvu
- GenSync obecné synchronizace
- Relay přeposílání rámců (vyjma rámců RT 3. třídy a synchronizačních)
- IR-FW přeposílání rámců RT 3. třídy (u zařízení s více porty)
- SY-FW přeposílání synchronizačních rámců (u zařízení s více porty)
- LLDP protokolu IEEE P802.1AB

DMPM používá pro mapování DMPM funkcí na Spojovou vrstvu její službu:

- MA_UNITDATA - pro přenos pomocí protokolu Spojové vrstvy (Ethernet rámců)

Kromě výše uvedených služeb a protokolů využívá Profinet IO služeb:

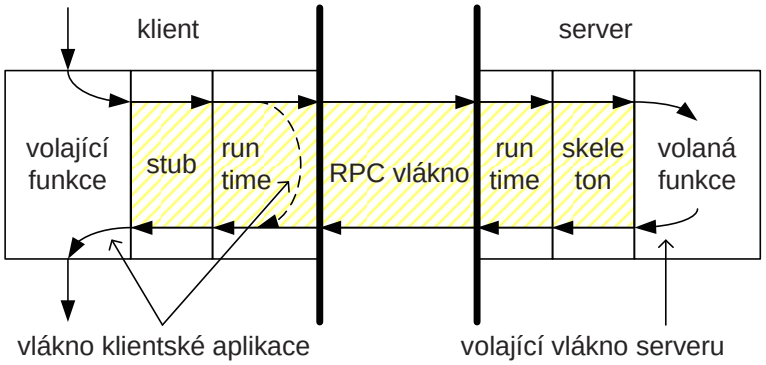
- RPC nespojovaného DCE RPC
- UDP IETF UDP protokolu za účelem přenosu CLRPC
- IP IETF IP protokolu za účelem přenosu UDP

CLRPC je využito pro přenos žádostí o služby pomocí mapovaných služeb:

- RPC_Connect - pro vytvoření aplikačního vztahu typu IO, nebo supervizor
- RPC_Release - pro zrušení aplikačního vztahu typu IO, nebo supervizor
- RPC_CControl - pro oznámení kontroléru, že je aplikace připravena
- RPC_DControl - pro oznámení zařízení, že jeho konfigurace byla dokončena
- RPC_Read - pro čtení dat ze zařízení
- RPC_Write - pro zápis dat do zařízení
- RPC_Alloc - pro alokaci paměti na vzdáleném systému

Remote procedure call

Techniku vzdáleného volání procedur (RPC) lze považovat, po přechodu od sekvenčně vykonávaných (děrné štítky), přes monolitické, k funkčně strukturovaným programům, za další změnu stylu programování. Vzdálené volání procedur je možno považovat za základní techniku tvorbu aplikací pro distribuované systémy.



Obr. 24, Vzdálené volání procedur

Princip této metody vychází a rozšiřuje lokálního volání funkcí, kde jsou parametry funkce uloženy na známé místo (většinou zásobník) a poté je zavolána požadovaná funkce, která si svá vstupní data z tohoto předem daného umístění načte, provede nad nimi svůj algoritmus a své výsledky poté uloží zase na známé místo (většinou registr), odkud si po návratu volající program výsledek vyčte. U vzdáleného volání procedur je situace obdobná, avšak není volána skutečná cílová funkce, ale jen tzv. funkce RPC rozraní, která předávané parametry zabalí do proudu dat a předá s dalšími parametry run-time knihovně k odeslání. Volající funkce je tomto místě do doručení odpovědi uspána a samotné volání dále pokračuje pouze v tzv. RPC vlákně. Proud parametrů je tak odeslán na cílový systém, kde je převzat run-time knihovnou serveru, která vytvoří volající vlákno v němž s z proudu vybalenými parametry zavolá skutečnou požadovanou funkci a její výsledky zase zabalí do

proudu dat a odešle nazpět klientovy stejnou cestou. Po doručení do run-time knihovny je proud vybalen a navrácen volající funkci jako výsledek a ta je v závěru probuzena z čekání na odpověď.

Pro tvorbu rozraní RPC se využívá speciálního jazyku Interface Definition Language (IDL), z kterého se za použití překladače IDL vygenerují hlavičkové soubory popisu rozhraní, zdrojový soubor klienta se stub funkcemi, které nahrazují exportované funkce a zdrojový soubor serveru, který implementuje tzv. skeleton, což je kód který volá podle požadavku ve vytvořeném vlákně exportovanou funkci.

Systémů postavených na tomto principu je dnes více, mezi nejrozšířenější je možno zařadit:

SUN RPC, které je běžně dostupné na systémech typu UNIX a je typické tím, že funkce exportovaného rozhranní mohou mít pouze jeden parametr, což se v praxi u více parametrických funkcí řeší jejich zabalováním do struktury, která je oním jediným parametrem.

- DCE/RPC, ze kterého vychází RPC používané na systémech typu Windows. Pro tuto verzi je naopak typické, že na funkce rozhranní nejsou kladeny téměř žádná omezení a dají se tak exportovat většinou přímo bez úprav což vyžaduje od překladače IDL popsání struktury zásobníku, na nějž jsou parametry funkce běžně ukládány.
- Systémy RPC jsou obecně poměrně rozsáhlé, neboť musí řešit mnoho problémů mezi které patří různé reprezentace dat, bezpečnost, přenos proměnných, navazování i samotná komunikace, která může být v závislosti na přenosovém kanálu buď spojovaná nebo nespojovaná.

Z uvedeného principu předávání parametrů vyplívají základní omezení programátora:

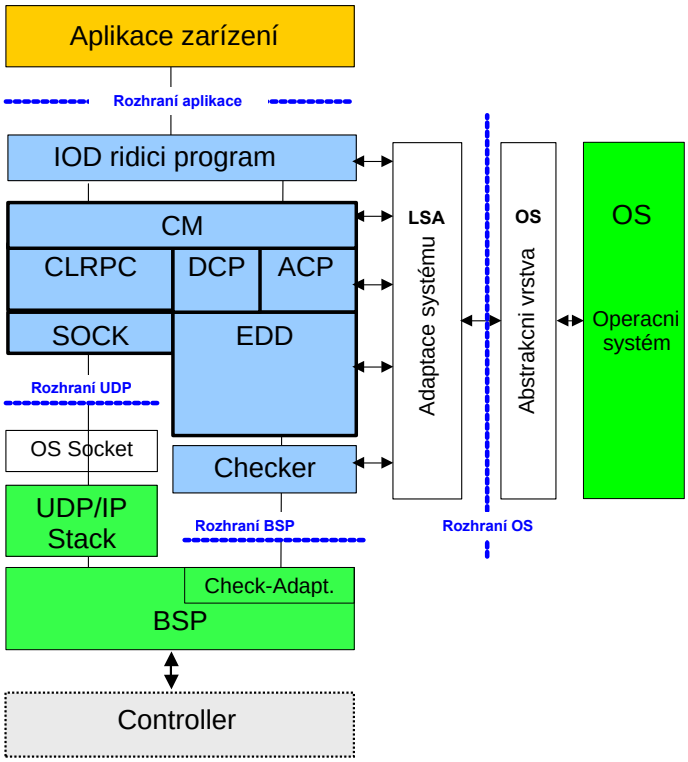
- Distribuovaná funkce nesmí používat parametry, které nejsou obsaženy v jejím rozhraní
- Při použití předání parametrů pomocí ukazatelů na data, musí být tato data dereferována, přibalena a odeslána s ostatními parametry, aby cílová funkce mohla nad těmito daty vykonat korektně svůj algoritmus.
- Paměťový prostor klienta i serveru jsou oddělené a s cela rozdílným obsahem, nelze tak využívat ukazatele na data v paměťovém prostoru protistrany.
- Jelikož pointerem referovaný prostor musí být přenášen jako parametr, je vhodné omezovat tento prostor na nejmenší možný. Z důvodu efektivity je tak někdy vhodnější rozdělit takovéto funkční volání na několik dílčích s dílčími nároky na přenášený prostor.

GSDML

GSDML je jazyk založený na XML a normě ISO 15745 o popisu zařízení pro umožnění jeho integrace. Tento jazyk umožňuje funkční popis (charakteristika zařízení, možná nastavení a konfigurace, popis volně použitelných diagnostických a chybových kódů, atd.) zařízení pro jeho možnou integraci pomocí běžných integračních nástrojů jako je Step7. Pro Profinet IO tento jazyk i strukturu popis nesoucího souboru definuje *GSDML Specification for Profinet IO.*

Profinet IO stack

Pro tvorbu IO zařízení podle specifikace Profinetu IO je použit PNIO stack vyvinutý firmou Siemens. PNIO stack je aplikace realizující protokol podle ve shodě s jeho specifikací a je určen pro zabudování do uživatelské aplikace, která bude využívat komunikaci přes Profinet IO. Využitím tohoto přístupu je programátor ostíněn od nutnosti studia celé specifikace protokolu a vystačí si s dokumentací k dodávanému stacku, která popisuje rozhraní stacku, která jsou potřeba adaptovat pro použité prostředí a popis API stacku vzhledem k aplikaci, která jej bude využívat. Pro snadný začátek je dodáván stack i se vzorovou aplikací využívající všechny dostupné prostředky a která je doporučována pro základní kámen uživatelovy aplikačně specifické aplikace. Vše je důkladně popsané v přiložené *PROFINET IO Developer's Kit : User guide.*



Obr. 25, Struktura komunikačního stacku

Na obrázku je znázorněna struktura dodaného PNIO stacku a jsou naznačena rozhraní vůči jednotlivým částem prostředí kterých stack využívá nebo kterým slouží.

PNIO stack, ač splňuje požadavky specifikace, má od ní na první pohled odlišnou strukturu, která se skládá z funkčních modulů zajišťujících:

- EDD (Ethernet Device Driver) poskytuje mechanizmy příjmu a odesílání cyklických (CRT) a acyklických (ART) realtime a (NRT) nerealtime rámců
- ACP provádí zpracovávání alarmů
- CM (Context manager) navazuje a spravuje komunikaci mezi IO zařízením a IO kontrolérem. Požadavky na navázaní komunikace přicházejí do správce kontextu IO zařízení od asociovaného IO kontroléru skrze nespojované RPC zajišťované moduly UDP a CLRPC.
- CLRPC implementuje nespojovanou verzi RPC protokolu
- DCP zajišťuje přiřazování IP konfigurace zařízení podle jeho názvu
- SOCK zajišťuje příjem a zasílání UDP paketů pro správce kontextu

- OS Socket vytváří rozhranní mezi IP/UDP stackem operačního systému a SOCK modulem pro zasílání a příjem UDP paketů a IP konfiguraci.
- OS abstrakce systému především sjednocuje datové typy a volání funkcí OS abstrakční vrstvy ze všech modulů, které je využívají pod svými názvy
- OS abstrakční vrstva odstiňuje PNIO stack od skutečného API použitého operačního systému
- IOD řídící program vytváří implementaci instance IO zařízení a poskytuje k ní aplikaci jednoduché rozhranní
- Checker je obdobou OS Socketu pro Ethernetové rámce. Zajišťuje tedy odesílání a příjem Ethernet rámců, registraci k jejich odběru z ovladače síťové karty a povoluje používání multicast adres.

Vyžadované prostředky

Pro svoji správnou funkci vyžaduje PNIO stack:

- UDP stack pro NRT komunikaci
- možnost odesílání a příjmu Eth. rámců pro CRT a ART komunikaci, potřebuje tedy přístup přímo k ovladači Eth. zařízení nebo co nejblíže jemu
- standardní prostředky RT operačních systémů mezi které patří: vlákna, semafory a mutexy, fronty správ, časovače, dynamicky alokovatelná paměť aj.

Aplikace

Aplikace v kontextu PNIO stacku realizuje pouze funkcionalitu samotného zařízení, neboť jí PNIO stack odstiňuje svým uživatelským API od samotné komunikace a právě proto zůstává její tvorba na uživatelovi, který je však v počátcích podpořen dodanou vzorovou aplikací která obsahuje jednu instanci PNIO zařízení. Tato aplikace se je určena pro ovládání přes terminál na který současně vypisuje ladící a chybové hlášení. Skrze tento terminál je možno vyvolávat různé základní akce jako:

- aktivace/deaktivace cyklické výměny dat
- vkládání a vyjímání modulů zařízení
- vkládání a vyjímání submodulů zařízení
- zasílání procesních alarmů
- zasílání diagnostických alarmů
- zasílání alarmů vztažených k manipulaci s moduly a submoduly
- zobrazování aktuálních přenesených IO dat
- ostatní testovací funkce

Stack je poskytován ve formě zdrojových kódů v jazyce C takže může být z hlediska hardware a operačního systému přenesen na jakékoliv platformu.

Proto, aby takovéto přenášení bylo co nejjednodušší, využívá PNIO stack virtuálních rozhranní ke všem použitým skutečným rozhraním. Nezbytné adaptace jsou tak zapouzdřeny a sjednoceny do jednotlivých definovaných rozhranní přístupu k hardwaru a operačnímu systému, čímž se jednak ulehčí přenášení stacku upravováním na prostředí závislém kódu na jediném místě, ale také, jak je to zde použito, je možné vést evidenci zdrojů, čímž se usnadní kromě ladění také i správa těchto zdrojů, tj. např. uvolňování těchto zdrojů při ukončování činnosti.

```
#define MAXNUM_OF_TIMER      10            // maximální počet dostupných časovačů

typedef struct
{
    LSA_BOOL              Occupied;    // časovač je buď dostupný nebo již alokován
    LSA_BOOL              Running;     // časovač je buď spuštěn nebo zastaven
    struct timer_list     TxTimer;     // OS závislá paměťová realizace časovače
    LSA_VOID        (*callback_timeout)
                    ( LSA_UINT16 timer_id, LSA_USER_ID_TYPE user_id);
                                         // ukazatel na callback funkci přidělené
časovači
    LSA_UINT32            par;         // parametr předávaný callback funkci časovačem
    LSA_USER_ID_TYPE      user_id;     // ID uživatele
    LSA_UINT32            timer_base; // časová báze - 1/10/100 msec
    LSA_UINT32            timer_typ;
    LSA_UINT32            TimeVal_ms;        // časový úsek v msec
    LSA_UINT32            TimeVal_Ticks;     // časový úsek v ticích OS
    LSA_UINT32            ErrCnt;            // čítač chyb
} OS_TIMER;

static OS_TIMER           OsTimer [MAXNUM_OF_TIMERS];
```

Obr. 26, Příklad realizace evidenční struktury virtuálního rozhraní

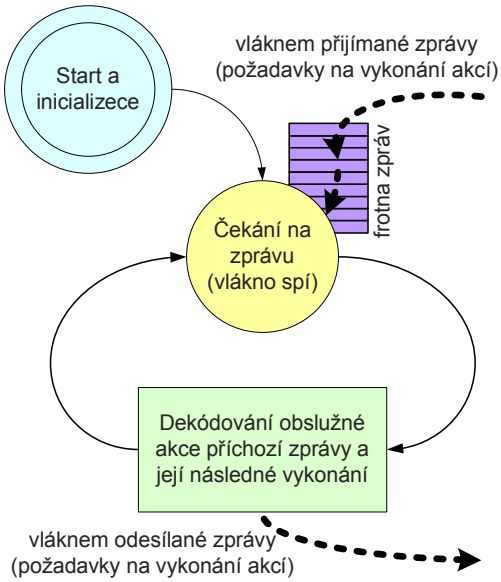
Organizace

Jak již bylo zmíněno u popisu standardu Profinet IO, je tento protokol po stránce obsahové specifikován definovanými ASEy a po stránce chování definovanými stavovými automaty generujícími protokol výměny APDU. Každé zařízení musí implementovat minimálně 10 základních definovaných stavových automatů: FSPMDEV, CMDEV, RMPM, PPM, CPM, ALPMR, ALPMI, APMS, APMR a LMPM kromě stavových automatů dalších zařízením použitých protokolů.

Specifikace Profinetu, jak vychází z konceptu ISO/OSI udává pouze realizaci abstraktního systému, který se může lišit od reálného systému svojí vnitřní organizací, nikoliv však vnějším chováním. Tohoto je využito u implementace Profinet IO stacku, který je tak naprogramován po vzoru událostmi řízených koordinovaných více vláknových aplikací. U tohoto návrhu se vychází z předpokladu, že jsou-li všechny stavové automaty realizovány ve svých vláknech je příliš mnoho, a naopak všechny v jediném je příliš málo. Hodně vláken totiž vytváří velikou administrativní zátěž na operační systém a koncept vše v jediném vlákně, pominu-li modularitu a odladitelnost, postrádá zbytky výhod preemptivního plánování.

Každý stavový automat je totiž rozepsán do akcí – obslužných procedur, které jsou-li zavolány vygenerují na základě svého vnitřního stavu a příchozí události stav následující a popřípadě vyvolají další událost zaslání žádosti o tuto akci příslušnému vláknu. Samotné nosné vlákno tak pouze čte zprávy jemu zaslané – žádosti o akce a tyto neprodleně vykonává, čímž v podstatě realizuje kooperativní multitasking. Nad tímto kooperativním multitaskingem, který může být navíc podpořen prioritním řazením zpráv v příchozích frontách jednotlivých vláken, je realizovaný primární – preemptivní multitasking plánovačem operačního systému.

Vhodným návrhem struktury aplikace, zejména rozdělením jednotlivých stavových automatů do skupin realizovaných jednotlivými vlákny lze dosáhnout optimálního výkonu, kdy jsou minimalizovány nároky na správu množství vláken, ale i minimalizované zpoždění vykonání akcí prioritních stavových automatů jejich vyčleněním do zvláštních vláken s nastavenou vysokou prioritou.



Obr. 27, Princip fungování obslužného vlákna u událostmi řízené aplikace

```
static PNIO_INT32 PNIO_Task (void)
{
    PNIO_UINT32     ThisTaskId = INVALID_TASK_ID;  // pro uložení ID vlákna
    void*           pRqb = NULL;                   // ukazatel na zprávu se žádostí o akci
    void*           pFct = NULL;                   // ukazatel na obslužnou rutinu

    OsWaitOnEnable();                              // čekání na povolení rozběhu
    ThisTaskId = OsGetThreadId ();                 // vyžádání své identifikace

    while (1)                                      // běh v nekonečné smyčce
    {
       /* vlákno čeká ve své blokující frontě na příchod zprávy */
        OsReadMessageBlocked (&pFct, &pRqb, ThisTaskId);

       /* ochrana před chybou segmentace paměti v případě vadné zprávy */
        if ((pFct == NULL) || (pRqb == NULL))
        { /* vyvolá pouze varovné hlášení pro vývojáře - chyba v programu */
            PNIO_ELOG (("Error PNIO_Task()\n"));
        }
        else
        { /* zavolání obslužné rutiny na příchozí zprávu se žádostí o akci */
            ((void(*)(void*))(pFct))((void*)pRqb);
        }
    }
}
```

Obr. 28, Příklad realizace obslužného vlákna u událostmi řízené aplikace

Vývoj pro ThreadX

Vývojová deska NetSilicon Net+50

Vývojová deska NetSilicon Net+50 je poskytována pro podporu vývoje aplikací založených na procesoru Net+50 vyráběného firmou NetSilicon. Tento procesor je postaven na rozšířeném licencovaném 32b RISC jádře ARM7TDMI s podporou DMA, které však bohužel postrádá podporu stránkování paměti, což zužuje výběr použitelných operačních systémů. Procesor je však vybaven jedním rozhraním MII pro 10/100Base-T Ethernet.

K vývojové desce je dodávána distribuce operačního systému NetOS firmy NetSilicon založeného na jádře ThreadX od firmy Express Logic. Jádro tohoto operačního systému je založeno na pikokernelové architektuře se zacílením na nasazení v hard realtime vestavěných zařízeních, tomu odpovídá i jeho determinističnost, rychlost obsluhy přerušení a preemptivní plánovač. Díky svému cílovému segmentu postrádá jádro podporu vizualizace paměti a dynamického linkování v čehož důsledku tak aplikace sdílí s jádrem jeden adresní prostor a spolu s jádrem se překládá do jediného monolitického obrazu. Pro snadnou tvorbu aplikací poskytuje ThreadX bohaté API se širokou podporou v oblasti vláken, časovačů, front zpráv, správy paměti, semaforů a dalších rozšířených synchronizačních nástrojů.

Celá distribuce se skládá z jádra operačního systému ThreadX ve formě předkompilované a optimalizované C knihovny pro použitý procesor spolu s definicí jeho API v hlavičkovém souboru tx_api.h. Další součástí je tzv. Board Support Package (BSP), což je sada ovladačů a jiného platformě závislého kódu napsaného speciálně pro použitý procesor a desku. Pro rozšíření jádra OS o další funkce jakými jsou např. TCP/IP stack a nad ním postavené aplikační protokoly, podpora USB a různých souborových systémů, různé šifrovací algoritmy a mnohé jiné, je poskytována sada zdrojových kódů vystavěných nad jádrem ThreadX. Použitá vývojová deska zahrnuje také vývojové prostředí postavené na GNU nástrojích v prostředí Cygwin, jinak je i dodávaná verze na základě proprietárního vývojového prostředí.

Obě tato prostředí jsou tak zacílená na uživatele operačního systému Windows. Vývoj pod operačním systémem GNU/Linux je možný za předpokladu, že si vývojové programy používané pod Cygwin znovu přeložíme pod Linux, přičemž se nesmí opominout, že tyto nástroje sice poběží pod Linuxem na našem PC, ale budou pracovat s kódem pro procesor na vývojové desce (tzv. křížové – cross nástroje).

Z důvodu absence sekundárního typu bootloaderu typu U-boot probíhá testování aplikace u této desky jejím nahráním buď do Flash nebo RAM paměti pomocí JTAG adaptéru s případnou možností následného trasování pomocí nástroje gdb.

Součástí vývojové sady je jak příručka pro rychlé zprovoznění vzorové aplikace, tak i sada zevrubné dokumentace nezbytné pro následný vývoj.

Dodaný PNIO stack je dodáván s již přenesenými rozhranní mi na NetOS a použitou vývojovou desku, čímž sice odpadají úpravy rozhranní, jak je zevrubně popisuje uživatelská příručka ke stacku, ale použitím překladače GNU GCC namísto u stacku použitého proprietárního překladače vyžaduje následující zásahy:
vytvoření nového Makefile s popisem závislostí pro překlad

- poupravení konfiguračního souboru v compiler.h

- ošetření rozdílnosti atributů překladačů pro pakování struktur
- odstranit varování a chybová hlášení vystavené překladačem
- zkontrolovat a případně poupravit velikost písmen v názvech souborů

V případě použití jiné verze NetOS než je 5.0 nebo 6.0, pro které dodává Siemens záplatu na BSP realizující přemostění TCP/IP stacku vytvořením druhé, prioritní, fronty pro odesílání Ethernetovských rámců a také rozhraní pro registraci filtračních funkcí aplikovaných na přicházející rámce. Právě z takovéto filtrační funkce dostává PNIO stack své Ethernetové rámce a právě skrze onu prioritní frontu poté své rámce odesílá.

NetOS verze 6.2 a výše již mají tato rozšíření implementována a dodaná záplata se sice nepoužije, ale v důsledku rozdílné implementace pomocí původní záplaty a rozhraní pozdějších verzí je nutná úprava na straně PNIO stacku v enic.c

Vytvoření nového Makefile

Původní kompilátor pro překlad PNIO stacku si generuje závislosti pro překlad automaticky. U GNU GCC není tato funkce přímo implementována a předpokládá se, že ji buď programátor u malých softwarových projektů buď nevyžaduje, nebo si ji sám vytvoří pomocí skriptu a Makefile s využitím např. GNU GCC s přepínačem –M, který pro zadaný zdrojový soubor vygeneruje jeho závislosti.

Já jsem se vydal cestou využívání projektu Makedep z Sourceforge.org <http://sourceforge.net/projects/makedep>. Jedná se generátor závislostí pro velké projekty v jazyce C/C++. Zanalyzuje všechny zdrojové a hlavičkové soubory v adresářovém stromě a vytvoří soubory se seznamem zdrojových souborů k překladu, jejich závislosti a seznam adresářů kde jsou obsaženy hlavičkové soubory. Jedná se o dva skripty v jazyce Perl, čímž je zajištěna jejich přenositelnost mezi platformami. První *mkfiles.pl* provede indexaci všech souborů v aktuálním adresáři a jeho podadresářích a seznam nalezených zdrojových souborů uloží do souboru *make.files*, který je využíván druhým skriptem *mkdep.pl* pro tvorbu dvou souborů vkládaných do univerzálního *makefile*. První (make.dep) obsahuje nalezené závislosti a druhý (make.ofiles) seznam vytvářených objektových souborů a seznam cesty pro hledání souborů překladačem.

```
PROGRAM_NAME    = tst          # jméno vytvářeného programu (v případě zvolení tohoto cíle)
PROGRAM_EXT     = .exe         # přípona vytvářeného programu
LIBRARY_NAME    = libtst       # název vytvářené knihovny   (v případě zvolení tohoto cíle)
CC              = gcc          # jméno a cesta k použitému překladači
AR              = ar           # jméno a cesta k použitému ar

all : $(PROGRAM_NAME)$(PROGRAM_EXT)  # cíl pro tvorbu programu

lib : lib$(LIBRARY_NAME).so          # cíl pro tvorbu statické knihovny

-include make.dep                    # import vygenerovaných závislostí
-include make.ofiles                 # import vygenerovaných objektů a cest


$(PROGRAM_NAME)$(PROGRAM_EXT) : $(OBJECTS)  # cíl pro slinkování programu s jeho objektů
        $(CC) -o $@ $(OBJECTS) $(INCLUDEPATH:-I=)

lib$(LIBRARY_NAME).so : $(OBJECTS)   # cíl pro slinkování statické knihovny z jejích
objektů
        $(AR) -r $@ $(OBJECTS)

%.o : %.c                            # generické pravidlo pro tvorbu objektů ze souborů *.c
        $(CC)-o $@ -c $< $(INCLUDEPATH:-I=)


.PHONY : depend
```

```
depend :                            # cíl pro tvorbu závislostí
        ./mkfiles.pl
        ./mkdep.pl

.PHONY : mrproper
mrproper : clean                    # cíl pro kompletní vyčištění projektu
        rm -f make.dep make.files make.ofiles

.PHONY : clean
clean  :                            # cíl pro vyčištění projektu
        find -name '*.o' -print | xargs rm -f
        rm -f $(PROGRAM_NAME)$(PROGRAM_EXT)
        rm -f lib$(LIBRARY_NAME).so
```

Obr. 29, Příklad generického makefile využívajícího skripty projektu MakeDep

Při tvorbě aplikace pro NetOS, která je spolu operačním systémem integrována do jediného obrazu, jsme nuceni využít jeho build systému ve formě hierarchického systému Makefile souborů. Makefile volaný pro překlad aplikace tak jen popisuje použité zdrojové soubory, knihovny a další volby a sám s tímto popisem aplikace volá nadřazené soubory Makefile, aby podle jeho popisu aplikaci přeložily a slinkovaly dohromady s předem vytvořeným obrazem operačního systému.

Vzorový makefile je nutno rozšířit a upravit následujícím způsobem:

```
-include make.dep                   # import vygenerovaných závislostí
-include make.ofiles                # import vygenerovaných objektů a cest

…

APP_C_FILES = $(OBJECTS:.o=.c)      # seznam zdrojových souborů ze seznamu budoucích
objektů
APP_INCDIRS = $(INCLUDEPATH)        # seznam prohledávaných cest

…

.PHONY : depend
depend :                            # cíl pro tvorbu závislostí
        ./mkfiles.pl
        ./mkdep.pl
```

Velikost písmen v názvech souborů

Linux je case sensitive a *.c a *.C představují pro GCC rozdílné typy souborů. Zatímco *.c je souborem se zdrojovým kódem zapsaným v čistém C. Naopak se souborem s příponou *.C nakládá GCC jako se souborem obsahujícím zdrojový kód zapsaný v jazyce C++ tedy s objektovým nástupcem prostého C, který je mezi jiné rozšířen o přetěžování funkcí. Přetěžování funkcí je funkce překladače, umožňující deklaraci více funkcí téhož jména lišící se pouze v počtu a typu argumentů. Překlad takových to funkcí pro možnost jednoznačného slinkování objektových souborů je realizován tzv. komolením jmen funkcí, tj. každému názvu funkce je při překladu přidán sufix společně a jednoznačně identifikující funkce dané třídy s danými argumenty. V případě, že se tedy mezi soubory v jazyce C připlete soubor sice napsaný v jazyce C, ale podle chybné přípony považovaný překladačem za soubor v jazyce C++, skončí slinkování takto vygenerovaných objektových souborů chybou, neboť linker nenajde symboly funkcí, které sice v objektovém souboru jsou reálně jsou, ale se špatnými – zkomolenými názvy.

Zarovnávání struktur

U dnešních procesorů s 16, 32 a vícebitovou datovou sběrnicí se projevuje problematika zarovnaného a nezarovnaného přístupu do paměti u datových položek s různou délkou slova. Moderní procesory jsou totiž hardwarově optimalizovány pro přístup do paměti zarovnaném po velikosti přistupované datové položky. Tímto přístupem vzniká mezi položkami, kdy první je menší délky než druhá položka a zároveň první nezarovnává svojí

velikostí adresu pro přístup k druhé, k vytvoření zarovnávající mezery, dochází k tzv. vycpání (padding). V případě požadavku přístupu k paměti bez zarovnávání, musí toto hardwarové omezení vyřešit překladač softwarově např. skládáním datové položky po menších zarovnaných blocích. Běžně se používá princip zarovnané paměti, řešení přístupu k nezarovnávané paměti je zejména doménou dekódování datových hlaviček např. protokolů. Novější protokoly však již zpravidla využívají implicitně zarovnané datové hlavičky a řešení této problematiky pro ně tím pádem odpadá.

```
struct
{
    INT8     polozka1;    // velikost  8b, zarovnávaná po  8b
    INT16    polozka2;    // velikost 16b, zarovnávaná po 16b
    INT32    polozka3;    // velikost 32b, zarovnávaná po 32b
} nezarovnaná_struktura;
```

Obr. 30, Příklad nezarovnané struktury

```
struct
{
    INT8     polozka1;          // velikost  8b, zarovnávaná po  8b
    INT8     vycpávka;          // dorovnání do 16b
    INT16    polozka2;          // velikost 16b, zarovnávaná po 16b
    INT32    polozka3;          // velikost 32b, zarovnávaná po 32b
} implicitně_zarovnaná_struktura;
```

Obr. 31, Příklad implicitně zarovnané struktury

```
struct
{
    INT32    polozka3;          // velikost 32b, zarovnávaná po 32b
    INT16    polozka2;          // velikost 16b, zarovnávaná po 16b
    INT8     polozka1;          // velikost  8b, zarovnávaná po  8b
} implicitně_zarovnaná_struktura;
```

Obr. 32, Příklad implicitně zarovnané struktury

Jelikož se běžně používá zarovnaný přístup k datům, musí se překladači nějakým způsobem sdělit takovýto záměr. Obecně je toto možné dvěma způsoby, a to buď sdělit překladači, aby ke všem strukturám přistupoval nezarovnávaným způsobem, což v případě NetOS nepřipadá v úvahu, neboť nemáme možnost si sami přeložit tímto způsobem i jádro operačního systému dodávaného ve formě předkompilované knihovny. Aplikace by tak používala jiné uspořádání dat při komunikaci s jádrem přes jeho nezměněné API a tudíž by nefungovala.

Druhým způsobem je selektivní aplikace nezarovnávaného přístupu na vybrané struktury, které to vyžadují. Toto se děje pomocí atributů, u GCC je to atribut __attribute__ ((packed)), u ostatních překladačů je rozšířená skupina atributů #pragma pack(…) a #pragma noalign.

```
struct nuceně_zarovnaná_struktura
{
    INT8     polozka1;
    INT16    polozka2;
    INT32    polozka3;
} __attribute__ ((packed));
```

Obr. 33, Příklad nuceně zarovnané struktury u GCC

```
#pragma pack(1) /* zarovnání po: 1 = 8b, 2 = 16b, 4 = 32b */
struct nuceně_zarovnaná_struktura
{
    INT8     polozka1;
    INT16    polozka2;
    INT32    polozka3;
};
#pragma noalign
```

Obr. 34, Příklad nuceně zarovnané struktury u MS překladačů

Ve zdrojových kódech je při použití překladače GCC vyhledat podle atributů #pragma pack na zarovnání citlivé struktury a zkontrolovat jejich řádné zarovnání pomocí atributu GCC.

Úprava definicí v compiler.h

Se změnou kompilátoru může nastat potřeba změnit řady nastavení a tak jsou za tímto účelem tato nastavení soustředěna do souboru compiler.h. Je výhodné, aby se správná nastavené podle použitého kompilátoru a případně prostředí vybrala automaticky pomocí identifikačních proměnných prostředí, které překladač nebo operační systém za tímto účelem definuje.

```
#if defined __GNUC__
        /* GCC */
        #define TOOL_CHAIN_GNU_PPC          1
#elif defined WIN32
        /* Win95+ */
        #define TOOL_CHAIN_MICROSOFT        1      // for windows simulation dont set big
endian
#else
        /* elsewhere */
        #define TOOL_CHAIN_GREENHILLS_ARM   1
        //#define TOOL_CHAIN_GENERIC_32BIT   1
#endif
```

Jelikož vzhledem k rozdílné definici atributů pro nezarovnávaný přístup ke strukturám u nastavení TOOL_CHAIN_GREENHILLS_ARM je výhodnější použít nastavení pojmenované TOOL_CHAIN_GNU_PPC a pro ně odvodit v controler.h konfiguraci od zde uvedené konfigurace pro TOOL_CHAIN_GREENHILLS_ARM s vynecháním neexitujícího hlavičkového souboru ghs_null.h.

Úprava stacku vhledem k BSP NetOS 6.3

Ethernet bypass API umožňuje aplikaci registraci k příjmu raw dat pomocí filtrovací funkce, čímž je zajištěna vysoká priorita vybraných vyfiltrovaných přijatých paketů a je tak umožněna bezprostřední odezva na vybrané typy.
Úprava stávající implementace založené na záplatách pro NetOS do verze 6.0 probíhají v souboru enic.c a potřeba:

- Přidat hlavičkový soubor API

```
#include "eth_stack.h"
```

- Odstranit zastaranou funkci ENIC_Isr_Recv()
- Zaměnit v celém souboru označení návratových stavů(nově jsou definovány ethFilterActionForward a ethFilterActionDiscard namísto zastaralých TELEGRAM_DISCARD a TELEGRAM_FORWARD)
- Zaregistrovat a odregistrovat filtrovací funkci

```
void ENIC_Open (void)
{
        .
        .
        .
// Registrace filtrovací funkce - ENIC_RecData
  NAEthRegisterCustomPacketFn(ENIC_RecData, NULL);
}

void ENIC_Close (void)
{
// Odregistrování filtrovací funkce - ENIC_RecData
  NAEthRegisterCustomPacketFn(NULL, NULL);

        .
        .
        .
}
```

- Nově implementovat funkci pro odesílání Eth. rámců

```
unsigned int Bsp_SndDatReq (void* pTxBuf, unsigned int Datlen, unsigned int Client)
{
    int localindex;
    m * new_packet;

    NA_UNUSED_PARAM(Client);

    if (pTxBuf != NULL)
    {
        localindex = ETH_FAILURE;
        int minBufferLength = (Datlen < MIN_9750_PACKET_SIZE) ? MIN_9750_PACKET_SIZE :
Datlen;
        if (pTxBuf && Datlen >= ETH_HEADER_LEN && Datlen <= ETH_MAX_PACKET_LENGTH )
        {
            /* Allocate a new packet from the stack */
            new_packet = NAEthMsgAllocate(minBufferLength);
            if (new_packet)
            {
                /* Copy data into the Ethernet buffer */
                fmemcpy(new_packet->m_hp, pTxBuf, Datlen);
                if (Datlen < minBufferLength)
                {
                    /*
                    * The Eth. transmitter on the BSP_ARM9 and NS9775 may stall if a packet
                    * less than 34 bytes is transmitted, so software must pad short
packets.
                    */
                    memset(new_packet->m_hp + Datlen, 0, minBufferLength - Datlen);
                }
            /* Send the packet out */
            /* Free function is NAEthMsgFree() */
            localindex = NAEthBypassTransmit(new_packet,NAEthMsgFree);
            }
        }
        if (localindex != ETH_SUCCESS)
        {
            return (LSA_NOT_OK);
        }

        return (LSA_OK);
    }
    /* error */
    return (LSA_NOT_OK);
}
```

Vývoj pro Linux

Vývojová deska FS Forth-Systeme A9M9750DEV

Vývojová deska FS Forth-Systeme A9M9750DEV je vyvinutá za účelem podpory vývoje aplikací založených na modulu CC9P9360_2 firmy Digi, která tento modul postavila na procesoru 9M9750. Tento procesor je postaven na výkonném licencovaném 32b RISC jádře ARM926EJ-S s podporou DMA a MMU, čehož důsledkem může být základem systémů s moderními operačními systémy jakými jsou Linux, WinCE a další. Na modulu je kromě procesoru integrovány paměti Flash a RAM jedno PHY připojené na rozhraní MII procesoru pro 10/100Base-T Ethernet.

K vývojové desce jsou dodávány distribuce operačního systému NetOS a Linux, ale i samotné BSP pro WinCE a Linux

Distribuce Linuxu pojmenovaná LxNETES je založena na vanilla linuxovém jádře 2.6.12.5 rozšířeném pomocí BSP, aplikaci BusyBox, bootloaderu U-Boot a křížovém vývojovém prostředí pro procesory ARM9. Za pomoci názorné příručky je instalace na doporučované distribuci Debian pro hostitelský počítač snadná. Součástí instalace je i nastavení TFTP serveru ze kterého si U-boot bootloader umí nahrát obraz operačního systému do paměti a následně spustit, což značně urychluje vývoj. Nakonfigurováním NFS serveru odpadá při vývoji potřeba přenášení obrazu souborového systému na vývojovou desku, neboť lze linux spustit v konfiguraci, kdy si je schopen připojit jako souborový systém adresář poskytovaný NFS serverem z hostitelského počítače

Linux

Linux je v pravém slova smyslu označením pouze jádra operačního systému, který bývá často zaměňován s pojmem distribuce operačního systému, tedy jak jádra, runtime knihoven, ale i všemožných aplikací. Operační systém ve své podstatě vytváří podporu pro běh uživatelských aplikací, nikoliv takovou aplikaci realizuje.

Z hlediska architektury systémů je Linux realizací monolitického jádra, pro které je specifické, že všechny aplikace uvnitř něj běží ve společném adresném prostoru. Toto naproti mikrokernelu, ke kterému směřují operační systémy Windows, nese s sebou větší požadavky na kvalitu kódu umísťovaného do jádra, protože tento může lehce ohrozit stabilitu celého systému, ale to je zase vyváženo možností snazší komunikace mezi jednotlivými částmi a tak i rychlejší odezvou, protože nemusí být všechny interakce jednotlivých částí realizovány zprávami, jako tomu je právě v koncepce mikrokernelu.

Z hlediska struktury Linuxu podává její nejlepší obraz samotná adresářová struktura zdrojového kódu. Linux je tak hlavně tvořen svým jádrem zajištujícím správu paměti, plánovač, podporu sítí a modulů. Za moduly jsou označovány v Linuxu část kódu, která běží v jádře a může být v závislosti na požadavcích vložena nebo vyjmuta z jádra obdobně jako se spouští a ukončuje uživatelská aplikace. Díky možnosti této dynamické manipulace s modulem je umožněno přidávat do jádra kód, aniž by bylo potřeba překládat znovu celé jádro, natož do něj dělat jiné zásahy. Tímto způsobem je umožněn autonomní vývoj minoritních projektů umístěných do jádra, kdy tyto se pak udržují a překládají samostatně.

Překlad aplikace

Překlad aplikace představuje standardní programátorskou činnost u větších projektů navíc prakticky vynuceným použitím nějakého programu pro řízení překladu, většinou programu

make. U takovýchto rozsáhlejších projektů je výhodou automatická tvorba závislostí mezi překládanými zdrojovými kódy. Z tohoto důvodu používám pro překlad skripty projektu MakeDep a generický makefile, jak byl představen v kapitole o vývoji pro ThreadX.
Při překladu aplikace pro linux běžící na vestavěném zařízení je také nutné při použití standardních knihoven požadovat jejich statické slinkování pomocí přepínače „–static“, neboť tyto distribuce linuxu zpravidla neobsahují dynamické knihovny.

Překlad modulu

Moduly jádra požadují jiný přístup k jejich překladu na rozdíl od aplikace. Musí být přeloženy oproti jádru pod kterým budou provozovány a tak je pro ně tím pádem vyžadováno i specifické nastavení kompilátoru a linkeru, už proto, že procesor nemusí umět hardwarově vykonávat operace v plovoucí řádové čárce, které pak operační systém musí pro aplikace softwarově emulovat.

Z tohoto důvodu se využívá při tvorbě makefile modulu build systém pro překlad samotného jádra, který nás tak od těchto specifických nastavění odstíní.

Naše makefile pak pouze popisuje strukturu zdrojových souborů využívaných pro tvorbu modulu, soupis cest, ve kterých má překladač defaultně hledat příslušné hlavičkové soubory a případná naše další nastavení. Tvorbu systému závislostí zdrojových souborů si poté již zajistní samotný vyvolaný build systém jádra.

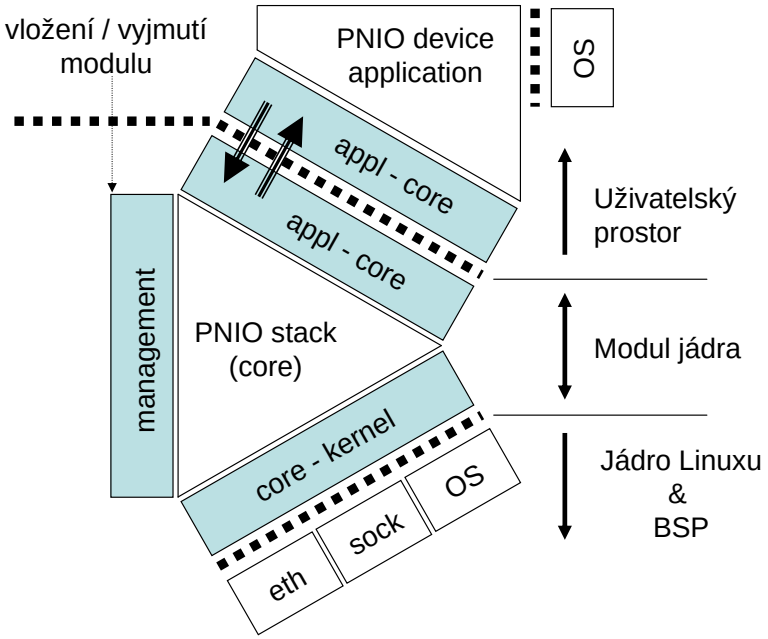
Makefile modulu :

```
ifneq ($(KERNELRELEASE),)

-include $(M)/make.dep
-include $(M)/make.ofiles

    EXTRA_CFLAGS += $(subst -I,-I$(M)/,$(INCLUDEPATH))
    obj-m        := module.o
    module-objs  := $(OBJECTS)

else
#*************************************************
    PWD         := $(shell pwd)
    KERNELDIR   ?= /lib/modules/$(shell uname -r)/build
    KERNELDIR_A := /root/LxNETES-3.2/build/linux/
#-------------------------------------------------
# verbose mode => make target V=1
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
        rm -f .*.cmd

arm:
        $(MAKE) -C $(KERNELDIR_A) M=$(PWD) modules ARCH=arm CROSS_COMPILE="/root/LxNETES-
3.2/bin/arm-linux-"
        cp module.ko /exports/nfsroot-cc9p9360dev
        rm -f .*.cmd
#-------------------------------------------------
.PHONY : depend
depend :
        ./mkfiles.pl
        ./mkdep.pl
#-------------------------------------------------
.PHONY : help
help   :
        $(MAKE) -C $(KERNELDIR) M=$(PWD) help
#-------------------------------------------------
.PHONY : clean
clean:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
#*************************************************
endif
```

Obr. 35, Příklad generického makefile pro překlad modulu

Nová architektura

Jak již bylo v úvodu nastíněno dochází k rozdělení komunikačního stacku mezi aplikační prostor a prostor jádra s čím je spojeno vytvoření rozhraní propojujícího oba tyto jinak oddělené světy. Součástí přenášení na novou platformu je zde samozřejmě i úprava rozhraní pro používání služeb jádra, socketu i Eth. rozhraní v jádře. Pro zachování konsistentního přístupu při programování uživatelské aplikace i nadále je vytvořeno rozhraní vůči OS i v uživatelském prostoru, které zajišťuje používané funkce a zavedené definice typů.



Obr. 36, Nové uspořádání komunikačního software

Rozhraní aplikace - jádro

Koncepce

Díky složitosti vývoje a údržby kódu psaného pro jádro, je tak běžně pouze jeho nejzbytnější a časově nejkritičtější část umísťovaná do jádra. Ostatní části, jako GUI, management a řízení jsou typické pro uživatelský prostor. Tato praxe rozdělování implementace některých částí mezi jádro a uživatelský prostor je běžná pro Linux.

Nezbytnou podmínkou pro realizovatelnost takovéhoto modelu implementace aplikace je schopnost propojit tyto dvě části aplikace umístěné navíc ve dvou zcela rozdílných prostředích pomocí nějakého druhu meziprocesní komunikace (IPC).

V linuxu existuje několik základních běžně používaných metod IPC pro propojení uživatelského prostoru a prostoru jádra:

- Systémová volání
- standardní souborové operace znakového zařízení
- IoCtl operace znakového zařízení
- /Proc souborový systém
- Obecné sockety
- Netlink socket

Bližší pohled na vyjmenované IPC:

- **Systémová volání**

Systémová volání jsou základem všech interakcí jádra s aplikacemi a pomocí nichž jsou tedy přirozeně realizovány i všechny ostatní metody. Obecně proces není schopen přistupovat do jádra. Nemůže přistupovat do jeho paměti a ni volat jeho funkce, což je způsobeno během tohoto procesu v tzv. protected módu. Jedinou možností řešení tohoto přístupu je tak využití zadních vrátek ve formě uložení kódu požadované akce jádra na definované umístění (registr procesoru) a vyvolání příslušného softwarového přerušení, čímž dojde k přechodu do prostoru jádra, kde je přirozeně dovoleno vše. Jádro podle parametru určí o jaké systémové volání jde a zavolá příslušející obslužnou rutinu, která aplikací vyžadovanou funkci vykoná. V praxi uživatelská aplikace nepoužívá přímo systémová volání, ale volá funkce ze standardní knihovny, která již zajistí uskutečnění systémového volání. Nevýhodou této nejuniverzálnější metody je, že není jednoduché přidat položku do systémových volání. Kód implementující systémová volání v jádře totiž vyžaduje statické slinkování, které je možné jen při kompilování jádra, takže nepřipadá v úvahu využití systémových volání v dynamicky vkládatelném modulu.

- **Standardní souborové operace znakového zařízení**

Pro užití této metody musí modul realizovat ovladač znakového zařízení, kterým je např. modem, který pouze průběžně zpracovává sériová data. U takovýchto zařízení poté můžeme do modulu přistupovat pomocí vybraných a námi implementovaných standardních souborových operací. Aplikace poté přistupuje k modulu jako k běžnému souboru, který je neskutečnosti pouhým inod vytvořeným za tímto účelem v adresáři /dev a svázaným pomocí hlavního a vedlejšího čísla a typu zařízení se svým modulem. Mezi základní funkce patří otevření, zavření, zápis a čtení do a ze zařízení implementovaného v modulu.

- **Ioctl (IO control)**

Ioctl je součástí standardních souborových operací a je předurčena jako sekundární komunikační kanál do ovladače zařízení za účelem jeho konfigurace a správy. Díky tomuto druhému kanálu může být primární kanál tvořený operacemi zápisu a čtení z a do zařízení použit pouze pro přenos skutečných dat. Volání Ioctl inicializuje uživatelská aplikace voláním příslušné funkce ve které adresuje zařízení pomocí deskriptoru inodu, který patří zařízení a dále nese Ioctl číslo a případná další data nebo odkaz na místo v paměti od kuť nebo kam může modul číst a zapisovat data. Ioctl číslo globálně adresuje zařízení a příkaz, který má vykonat. Toto číslo tak musí sdílet jak aplikace tak i modul.

- **Souborový systém /proc**

V Linuxu je ještě další způsob zprostředkovávání informací mezi modulem a aplikací, a to pomocí /proc souborového systému, původně vytvořeného za účelem jednoduchého přístupu ke statistickým informacím o operačním systému. Nyní se však již jedná o velmi rozšířený způsob reportování informací různými částmi jádra. Princip implementace je obdobný jako u znakového zařízení, ale není potřeba vytvářet komunikační soubor, který je zde vytvářen automaticky. Také je zaměřen pouze na jednorázové bezestavové interakce, kdy se na rozdíl od znakového zařízení nevytváří relace ve formě deskriptoru otevřeného souboru.

Systémová volání a Ioctl jsou jednostranné typy IPC v tom smyslu, že vyvolání IPC lze realizovat pouze ze strany aplikace v uživatelském prostoru, což je problém když chce modul v jádře poslat nějakou urgentní zprávu aplikaci. V tomto směru zde není možnost přímé realizace. U těchto přístupů by se tak musela aplikace periodicky dotazovat modulu,

zda pro ni má nějaké sdělení, což představuje zbytečné plýtvání zdroji a v případě vyžadované rychlé odezvy s ohledem na nutnou frekvenci těchto dotazů nereálné.

U realizaci založené na standardních souborových operacích znakového zařízení a nebo na souborovém systému /proc, lze požadavek zpětného kanálu realizovat využitím čtení z modulu jako blokované operace, která je v modulu uspána při čekání na zprávu pro uživatelskou aplikaci. Tímto je vytvořen požadavek, aby takovéto čtení v uživatelské aplikaci probíhalo ve vlastním čtecím vlákně, které je probouzeno až když modul předá zprávu aplikaci. Zápis dat do modulu probíhá standardně. Při této implementaci je nutné na straně aplikace vypnout vyhrazení současného přístup do tohoto souboru pouze jediným vláknem např. pomocí nestandardní funkce libc: __fsetlocking() zavolané s parametrem FSETLOCKING_BYCALLER, který libc sděluje, že si uživatel bude řídit přístup k souboru sám. Další možností je používání _unlocked variant standardních funkcí.

Pro všechny výše zmíněné způsoby platí, že se musí používat speciálních funkcí pro přenos dat mezi paměťovým prostorem jádra a paměťovým prostorem aplikace, jako jsou především copy_to_user() copy_from_user() a další.

Důvodem pro užití těchto funkcí je, že paměť v Linuxu je segmentovaná. Toto znamená, že ukazatel sám o sobě neukazuje na unikátní místo v paměti, ale pouze na unikátní místo v právě nastaveném segmentu. A my tak potřebujeme tedy potřebujeme navíc znát správný segment, abychom mohly přistupovat k požadovanému místu. Jádro linuxu je umístěno ve svém vlastním segmentu, stejně jako každá ostatní běžící aplikace (procesy). Tím pádem je ukazatel unikátním označením místa v paměti pouze pro svůj segment, což nám nemusí vadit, pokavaď provádíme paměťové přístupy pouze do prostoru svého procesu. Při psaní modulů jádra obvykle potřebujeme většinou přistupovat do paměti jádra udané segmentem paměti, který si systém sám přepíná. Nicméně v případě, že potřebujeme přenášet data mezi prostorem jádra a prostorem aplikace potřebujeme přepínat tento segment, což zajišťují např. výše uvedené funkce

- **Netlink**

Netlink je speciální IPC použitelné pro přenos informací mezi jádrem a aplikací v uživatelském prostoru. Netlink poskytuje plně duplexní propojení mezi uživatelským prostorem, kde aplikace využívá klasické socket rozhraní, zde pouze místo AF_INET skupiny adres užívaných pro TCP/IP přenos využívá svojí skupinu AF_NETLINK.

Naproti tomu modul v jádře využívá speciální rozhranní definované hlavičkovým souborem <linux/netlink.h>. Na rozdíl od systémových volání není u netlinku problémem přidat nový typ protokolu do netlink.h a poté hned může aplikace a modulem začít využívat toto soketově orientované API.

Netlink je asynchronní, protože jako ostatní soketové API zajišťuje fronty zpráv při výměně dat. Systémové volání pro odeslání netlinkové zprávy zařadí zprávu do fronty zpráv příjemce a poté je vyvolána obslužná rutina příjemce. Příjemce zprávy se poté může rozhodnout, zda si ji hned zpracuje sám ve svém kontextu, nebo ji zpracuje později v jiném kontextu. Na rozdíl od netlinku vyžadují ostatní způsoby synchronní zpracování. Takže například užití systémových volání pro předávání zpráv z a do jádra z uživatelské aplikace může při náročnějším zpracování tohoto volání v jádře narušit plánování. Tento problém lze samozřejmě vyřešit vytvořením front zpráv, ale to je právě první krok k použití netlinku. Na možnost budoucího využití netlink socketu u vkládatelného objektu tak není potřeba pamatovat při překladu jádra.

Netlink využívá standardní socketové rozhraní s použitím své skupiny adres AF_NETLINK, typ socketu může být buď SOCK_RAW nebo SOCK_DGRAM, kdy

chováním mezi nimi rozdíl. Skupina protokolů kde označuje využitý komunikační kanál (typ protokolu), standardně používané jsou v netlink.h, pro vlastní kanál si musíme nadefinovat vlastní.

Netlink podporuje až 32skupin pro multicast pro každý komunikační kanál. Adresace multicast skupin probíhá pomocí bitové masky. Adresa jedné takovéto skupiny pak daná bitovou maskou 1 << i , kde $0 \leq i \leq 31$. Toto je extrémně užitečné, pro koordinaci a synchronizaci řady procesů přiřazených do skupin, kdy tento systém rasantně redukuje rozsah a nároky na implementaci řešení takovéhoto systému.

Netlink socket je flexibilní rozhranní pro komunikaci mezi uživatelskou aplikací a modulem jádra. Poskytuje lehce použitelnou socketovou API pro aplikaci i modul. Poskytuje rozšířené možnosti komunikace, jako plnou duplexnost, bufferování I/O, multicast a asynchronní komunikací s vyloučením použití ostatních kernel- aplikace IPC.

Implementace rozhraní

Komunikace mezi uživatelskou aplikací a PNIO stackem probíhá skrze volání definovaných funkcí a callback funkcí. Za tímto účelem bylo vytvořeno rozhraní mezi aplikací a modulem na fungující na principu zjednodušeného nespojovaného RPC. Toto rozhraní využívá shodnost obou prostředí, čímž tak odpadají problémy spojené transformací mezi různými representacemi dat, což rozhraní značně zjednodušuje.
Implementace rozhraní dodržuje základní schéma kdy:

- Vlákno volající vzdálenou funkci volá ve skutečnosti stub funkci rozhraní
- Ve stub funkci dochází k serializaci vstupních argumentů
- Serializované argumenty jsou odeslány Netlink rozhraním na druhou stranu rozhraní a volající vlákno je ve stub funkci uspáno
- Přenesená data na druhé straně jsou deserializovaná ve skeleton funkci ze které je pak s těmito argumenty zavolána skutečná implementace vzdálené funkce
- Výstupní argumenty vzdálené funkce jsou opět serializovány a odeslány nazpět
- Navrácené výstupní argumenty jsou deserializovány do stub funkce
- Po deserializaci výstupních argumentů je volající vlákno probuzeno a opouští stub funkci s výstupními argumenty skutečné vzdálené funkce.

Princip serializace (balení argumentů do řetězce) a deserializace (vybalování argumentů z řetězce) je zde postaven na principu maker, kdy pro každý datový typ za tímto účelem vytvořena skupina maker toto pro něj zajišťující. Struktury jsou v tomto konceptu považovány pouze za seskupení v nich obsažených položek, které se kterými se tak nakládá makry podle jejich typů.

- definice operačních kódů přenášených funkcí

```
typedef enum{
//    FOO,
    opc_fce1,
    opc_fce2
} func_code;
```

- Makra pro de/serializaci položky elementárního typu (PNIO_INT32)

```
/* PNIO_INT32 */
#define VARIABLE_DEF_PNIO_INT32(pol)  PNIO_INT32  pol
#define SERIALIZE_PNIO_INT32(pol)     (*(PNIO_INT32*)ptr) = (pol), ptr+= sizeof(PNIO_INT32)
#define DESERIALIZE_PNIO_INT32(pol)   (pol) = (*(PNIO_INT32*)ptr), ptr+= sizeof(PNIO_INT32)
```

- Makra pro de/serializaci struktury - seskupení de/serializací jednotlivých položek

```
/* PNIO_DEV_ADDR*/
#define VARIABLE_DEF_PNIO_DEV_ADDR(pol)  PNIO_DEV_ADDR  pol
#define SERIALIZE_PNIO_DEV_ADDR(pol) \
                          SERIALIZE_PNIO_ADDR_TYPE((pol).Type),      \
                          SERIALIZE_PNIO_UINT32((pol).Geo.Slot),     \
                          SERIALIZE_PNIO_UINT32((pol).Geo.Subslot)
#define DESERIALIZE_PNIO_DEV_ADDR(pol) \
                          DESERIALIZE_PNIO_ADDR_TYPE((pol).Type),    \
                          DESERIALIZE_PNIO_UINT32((pol).Geo.Slot),   \
                          DESERIALIZE_PNIO_UINT32((pol).Geo.Subslot)
```

- Makra pro unifikovaný přístup

```
/* GLOBAL */
#define VARIABLE_DEF(type, pol) VARIABLE_DEF_##type(pol)
#define SERIALIZE(type, pol)    SERIALIZE_##type(pol)
#define DESERIALIZE(type, pol)  DESERIALIZE_##type(pol)
```

- Inicializační makra

```
/* INIT */
#define INIT_SERIALIZE()   char sbuff[BUFF_MAX_SIZE]; void *ptr = sbuff
#define INIT_DESERIALIZE() void *ptr = sbuff
#define INIT_RESERIALIZE() ptr = sbuff
```

- Makro pro odeslání serializovaných parametrů

```
/* SEND */
#define SEND_BUFF(func) ptr = stub_send(func, sbuff, ((int)ptr - (int)s   buff))
```

- Příklad stub funkce – serializační rutina na straně žadatele

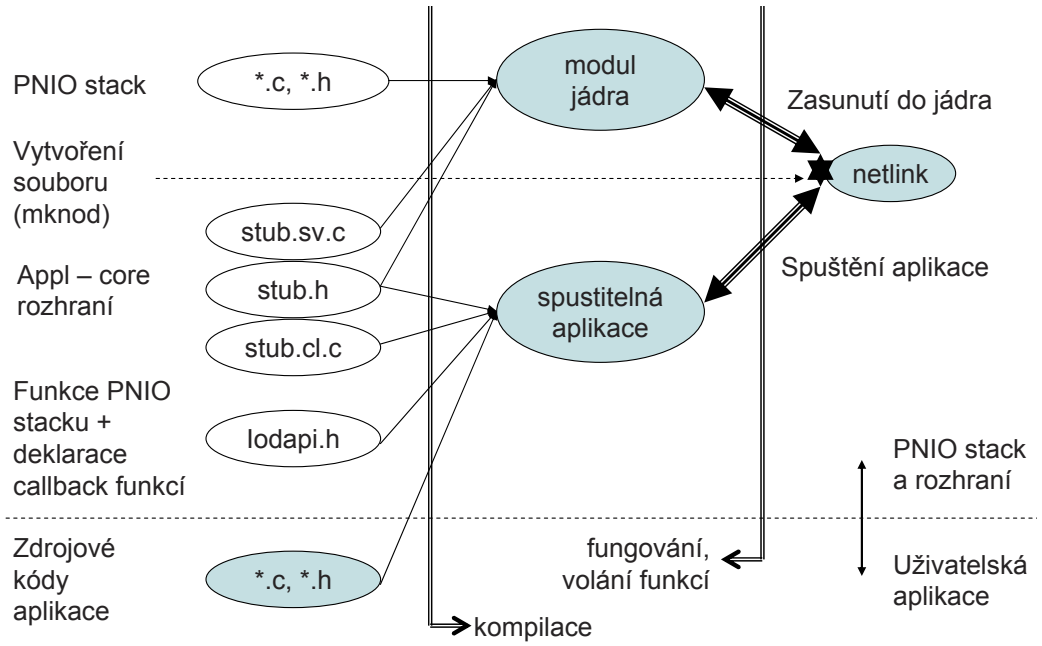
```
/* 6.1.1.1  fce1                    */
PNIO_UINT32  PNIO_fce1
             (PNIO_UINT32 par1,
              PNIO_UINT32 par2)
{
       INIT_SERIALIZE();
       VARIABLE_DEF(PNIO_UINT32, ret);      /* vytvoření lok. proměnné */

       SERIALIZE(PNIO_UINT32, par1);        /* vložení do řetězce   */
       SERIALIZE(PNIO_UINT32, par2);

       SEND_BUFF(opc_fce1);                 /* odeslání */

       DESERIALIZE(PNIO_UINT32, ret);       /* vyjmutí z řetězce   */
       return ret;
}
```

- Příklad serveru – deserializační část na straně volané funkce

```
PNIO_UINT32 stub_cl_recv(func_code func, unsigned int mutexId, void *context,
                                 char *sbuff)
{
       INIT_DESERIALIZE();

       switch(func)                                 /* výběr volané funkce */
       {
/* 6.1.2.1  PNIO_fce2     */
              case opc_fce2 :
              {
                  VARIABLE_DEF(PNIO_UINT32, par1); /* vytvoření lok. proměnné */
                  VARIABLE_DEF(PNIO_UINT32, par2);
                  VARIABLE_DEF(PNIO_UINT32, par3);

                  DESERIALIZE(PNIO_UINT32, par1);  /* vyjmutí z řetězce   */
                  DESERIALIZE(PNIO_UINT32, par2);  /* do lokální proměnné */
```

```
                DESERIALIZE(PNIO_UINT32, par3);

                PNIO_fce2(      par1,
                                par2,
                                par3 );

                INIT_RESERIALIZE();                /* příprava na serializaci */

                                                   /* void fce() - nic se nevrací
*/
            }; break;

/* DEFAULT -- error                        */
            default:
            {
                    return PNIO_NOT_OK;
            };
        } /* switch end*/

/* odeslání opovědi */
        stub_cl_send(TYPE_RSP, func, mutexId, context, ser_buff,
                            (int)ptr - (int)sbuff);

        return PNIO_OK;
}
```

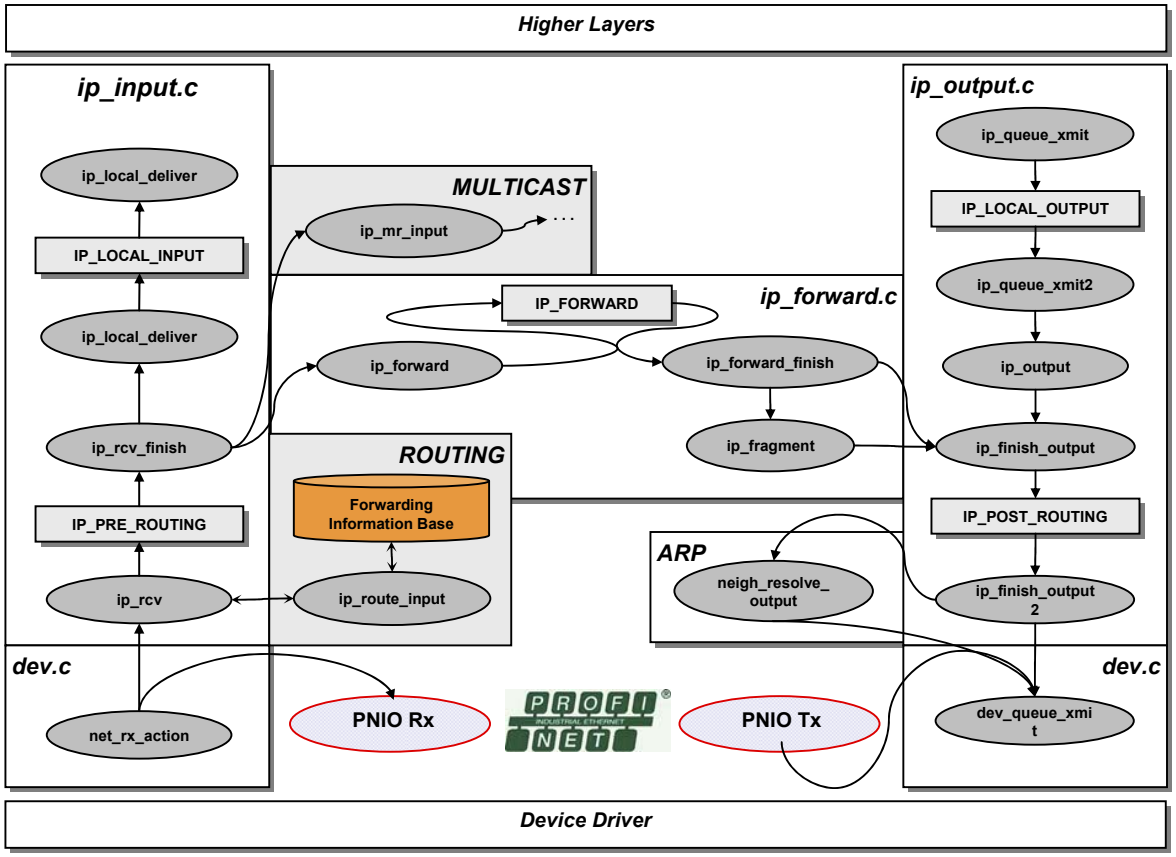
- Slinkování a vytvoření rozhraní



Obr. 37, Princip vzniku a fungování rozhraní

Rozhraní jádro – OS, Eth, Sock

Tato rozhraní jsou implementována na základě požadavků popsaných v Uživatelské příručce ke stacku a na základě funkcí popsaných v knize Linux Device Driver 3. edice (Corbet, Rubini, Kroah-Hartman) zdarma dostupné na internetu pod heslem LDD3.

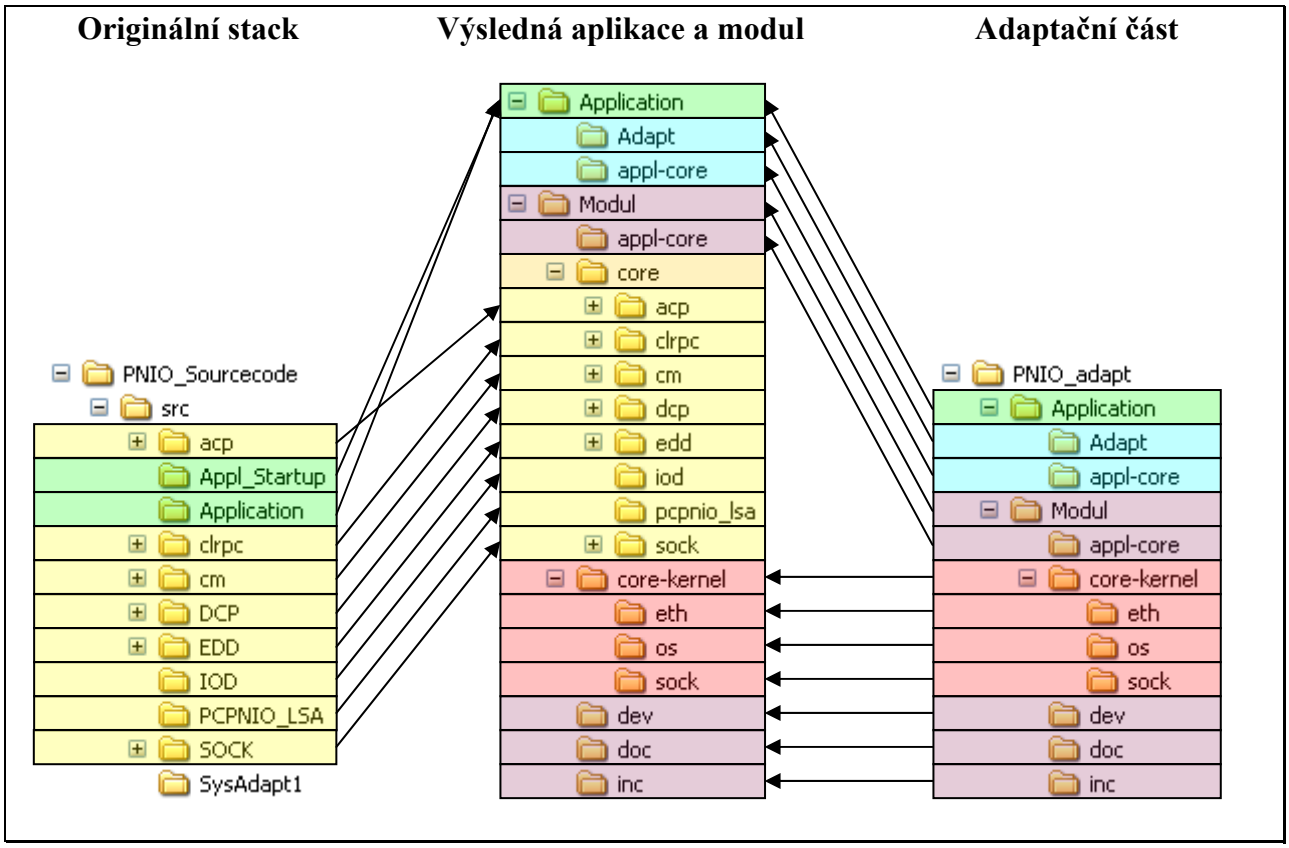


Obr. 38, Místo odchytu a odesílání Eth. rámců v relaci s TCP/IP stackem

Instalace PNIO stacku

Pro vývoj zařízení na PNIO stacku pod linuxem je nezbytné správně instalovat jak zdrojové kódy samotného PNIO stacku, tak i vytvořeného rozhraní. Za tímto účelem byla i uspořádaná adresářová struktura rozhraní, která je rozdělena na část pro aplikaci a část pro modul. Každá tato část je ve svém adresáři, ve kterém probíhá i překlad pomocí zde umístěných specifických Makefile souborů.

Pro dodaný stack je nezbytně nutné provést opatření z důvodu změny překladače, jak jsou uvedena v části pro ThreadX. Druhou nezbytnou úpravou je nutnost odstranění knihovních funkcí pokud se ve stacku vyskytují, popřípadě nekorektností použitých typů jako tomu je u verze PNIO v. 1.0.1.



Obr. 39, Sesazení upravených rozhraní a dodaného stacku do výsledné konfigurace

Závěr

V této práci jsem se zaměřil na představení Profinetu IO z jeho druhé strany, nikoliv z té funkční jak bývá často popisován, ale z hlediska abstraktního modelu jeho realizace zavedené ve schodě s Referenčním modelem OSI, který je v této práci taktéž představen.
Tato práce je zaměřena na vývoj zařízení na základě dodaného komunikačního stacku pro dva operační systémy a to ThreadX a Linux.
Vývoj pro ThreadX, jelikož s jeho rozhraním komunikační stack dodáván, nepředstavoval nutnost rozsáhlých zásahů, které byly vyvolány použitím jiného překladače a novější verze OS.
Vývoj pro Linux s sebou přinesl otázku umístění kódu, kdy byla zvolena varianta rozdělení implementace mezi prostor jádra a uživatelský prostor. Tato varianta byla z hlediska realizované implementace shledána za nejlepší přístup, ale vyžádala si vytvoření nové části kódu – rozhraní mezi těmito dvěmi světy a pro zachování stejného programátorského API i zdvojení úpravy rozhraní komunikačního stacku vůči službám OS jak pro prostor jádra, tak i pro uživatelskou aplikaci.

Literatura:

PROFIBUS International (2005). PROFINET IO - Application Layer Service Defition and Protokol Specifation Version 2.0 .
PROFIBUS Nutzeorganization e.V.

Popp, M.,Weber, K. (2004). The Rapid Way to PROFINET.
PROFIBUS Nutzeorganization e.V.

PROFIBUS International (2005). GSDML Specification for PROFINET IO Version 2.0.
PROFIBUS Nutzeorganization e.V.

IEEE Standards, [online], 2006:
<http://standards.ieee.org/getieee802/portfolio.html>

IEC. Real-time Ethernet PROFINET IO.
<http://www.iec.ch>

Technical Standard, DCE 1.1: Remote Procedure Call
<http://www.opengroup.org/onlinepubs/9629399/toc.htm>
<http://www.opengroup.org/onlinepubs/9629399/toc.pdf>

Corbet J., Rubini A., Kroah-Hartman G.: Linux Device Drivers, Third Edition
<http://lwn.net/Kernel/LDD3>

ISO 7498-1:1994 Information technology – Open Systems Interconnections – Basic Reference Model: The Basic Model
ekvivalentní :
ITU-T Recommendation X.200 (07/94) INFORMATION TECHNOLOGY - OPEN SYSTEMS INTERCONNECTION - BASIC REFERENCE MODEL: THE BASIC MODEL

ISO 7498-2:1989 Information technology – Open Systems Interconnections – Basic Reference Model – Part 2 : Security Architecture
ekvivalentní :
ITU-T Recommendation X.800 (1991) SECURITY ARCHITECTURE FOR OPEN SYSTEMS INTERCONECTION FOR CCITT APPLICATIONS

ISO 7498-3:1997 Information technology – Open Systems Interconnections – Basic Reference Model: Naming and addressing
ekvivalentní :
ITU-T Recommendation X.650 (10/96) Information technology – Open Systems Interconnections – Basic Reference Model: Naming and addressing

ISO 7498-4:1989 Information technology – Open Systems Interconnections – Basic Reference Model – Part 4: Management fremework
ekvivalentní :
ITU-T Recommendation X.700 (09/92) MANAGEMENT FRAMEWORK FOR OPEN SYSTEMS INTERCONNECTION (OSI) FOR CCITT APPLICATION