České Vysoké Učení Technické v Praze
Fakulta Elektrotechnická
Katedra Řídicí Techniky

DIPLOMOVÁ PRÁCE

Komunikace pro vestavěné aplikace

František Kalenský únor 2003

Prohlášení

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

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

František Kalenský

Poděkování

Na tomto místě bych rád poděkoval vedoucímu diplomové práce Dr. Ing. Zdeňkovi Hazálkovi za podněty, připomínky a cenné rady, které mi poskytl. Dále bych poděkoval společnosti UniControls a.s. za poskytnutí hardwaru a rad ing. Františka Moravce při oživování komunikačního modulu.

V neposlední řadě bych také poděkoval celé mojí rodině za finanční a morální podporu po celou dobu studia na ČVUT.

Abstrakt

Tato diplomová práce je věnována real-time operačnímu systému OSEW, který byl speciálně navrhnut pro vytváření řídících aplikací pro řídící jednotky v automobilech. Práce nejprve popisuje komunikační standardy založené na ISO/OSI modelu. Dále popisuje sériový komunikační protokol CAN, který je operačním systémem OSEW plně podporován pro komunikaci mezi řídícími jednotkami v automobilu. Práce zejména popisuje způsob vytváření OSEW aplikace ve vývojovém prostředí CodeWarrior a OSEW Builder na procesoru Motorola řady HC08. Nakonec ukazuje realizaci několika vzorových OSEW aplikací.

Abstract

This dissertation is dedicated to real-time operating system OSEW. The system was especially designed for creating of control applications for control units used in cars. Dissertation describes communication standards based on ISO/OSI figure. Farther it describes serial communication protocol CAN what is fully supported by operating system OSEW for communication between control units in car. The dissertation particularly describes the way of creating OSEW application in development environment CodeWarrior and OSEW Builder by means of microprocessor Motorola HC08. It shows implementation of some exemplary OSEW applications at the end.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato diplomová práce se zabývá real-time operačním systémem OSEK. Tento operační systém byl speciálně navržen pro použití v automobilovém průmyslu. Používá se pro vytváření řídících aplikací do řídících jednotek v automobilu. Byl navržen tak, aby mohl být implementován do "malých" procesorů, tzn. aby nevyžadoval velké hardwarové nároky. Dnešní moderní automobily jsou doslova nabité takovýmito řídícími jednotkami, které se starají o správnou funkci všech zařízení v automobilu. Všechny tyto řídící jednotky jsou mezi sebou propojeny pomocí sběrnice CAN (*Controller Area Network*) a právě tuto sběrnici používá operační systém OSEK, resp. jeho část zajišťující komunikaci - COM (*Communication*), k vytvoření komunikace mezi jednotlivými řídícími jednotkami, na kterých jsou spuštěny OSEK aplikace, zajišťující správnou činnost dané řídící jednotky (např. řídící jednotka pro klimatizaci, atd.). Takovéto řešení pak zajišťuje snadné propojení mezi jednotlivými senzory, čidly a řídícími jednotkami.

Tato práce právě v kapitole 5 ukazuje vývoj takovéto OSEK aplikace. Jako cílový hardaware nám byla zapůjčena deska s procesorem Motorola řady HC08AZ60 od společnosti UniControls a.s. Z tohoto důvodu bylo vybráno vývojové prostředí pro procesory Motorola CodeWarrior od společnosti Metrowerks, která je součástí splečnosti Motorola. Od této společnosti rovněž pochází i konkrétní operační systém OSEKturbo. Pro vytvoření definice OSEK aplikace v tzv. OIL jazyce (OSEK Implementation Language) byl použit vývojový nástroj OSEK Builder rovněž od společnosti Metrowerks. Výsledkem je pak řada OSEK aplikací uvedených v kapitole 6.

Práce rovněž obsahuje podrobný popis sériového komukačního protokolu CAN, jimž se zabývá v kapitole 3. Následující kapitola 4 je věnována právě opračnímu systému OSEK, jeho způsobu plánování, zpracování přerušení, použití alarmů, událostí, obluze chyb, ladění atd. Dále obsahuje popis komunikace mezi řídícími jednotkami v síti a mezi jednotlivými úlohami.

Kapitola 2 je věnována komunikačním standardům, zvláště pak referenčnímu ISO/OSI modelu a popisu jednotlivých vrstev modelu.

Součástí práce je i CD-ROM, na kterém je možné nalézt vzorové OSEK aplikace, dokumentaci k opeačnímu systému OSEK, sériovému protokolu CAN a dokumentaci k použitému hardwaru. Dále je zde možné nalézt prezentaci k vývojovému prostředí CodeWarrior a heply k použitým vývojovým prostředím.

Kapitola 2

Komunikační standardy

Materiály použité v této kapitole jsou čerpány z [1] a [2].

2.1 Referenční OSI model

K zajištění podmínek vzájemné komunikace mezi stanicemi přes obecnou propojovací síť je zapotřebí stanovit tzv. pravidla hry. V minulosti nastávaly problémy při komunikaci mezi zařízeními od různých výrobců právě z důvodu vzájemné nekompatibility komunikačních protokolů. Tento důvod vedl ke stanovení určitých požadavků na komunikaci mezi zařízeními. Tato doporučení v obecné podobě doporučují způsob komunikace mezi zařízeními od úrovně fyzického připojení až po způsob pohybu rámců v síti.

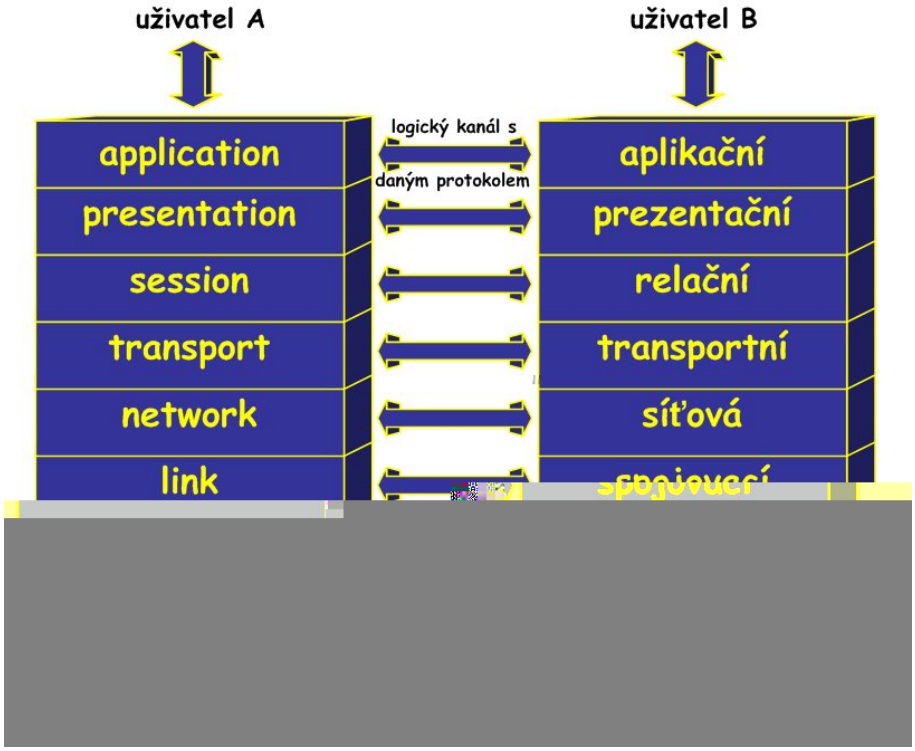
Doporučení pro vzájemnou komunikaci mezi zařízeními až do úrovně specifikace přenosu dat nebo programů vydala organizace *International Standard Organization* (ISO) a označila ji jako *Open system Interconnection* (OSI). Model OSI (viz obrázek 2.1) popisuje komunikaci mezi zařízeními jako hierarchii sedmi vrstev technických a programových prostředků, kde každá z vrstev zajišťuje funkce potřebné pro vrstvu vyšší a využívá služeb vrstvy nižší. Mezi jednotlivými vrstvami jsou (formou standardů a doporučení) definována rozhraní (*mezivrstvové protokoly*). Mezi prvky stejné vrstvy jsou definována pravidla komunikace (*vrstvové protokoly*).

2.1.1 Fyzická vrstva

Fyzická vrstva definuje fyzické propojení jednotlivých prvků zapojených v síti, tj. mechanické vlastnosti (konektory, typ a vlastnosti použitého přenosového média atd.), elektrické vlastnosti (napěťové úrovně, způsob kódování a modulace), dále může také definovat metodu přístupu k přenosovému médiu.

2.1.2 Spojovací vrstva

Spojovací vrstva můžeme rozdělit na dvě podvrstvy, na vrstvu řídících logických spojů (*Logical Link Control* (LLC)) a vrstvu zajišťující přístup na médium (*Medium Access Control* (MAC)). Díky tomuto oddělení může jedna logická vrstva LLC využívat několik různých vrstev MAC. Spojovací vrstva zajišťuje data proti chybám při přenosu, dále se stará o přenos zpráv v síti, které jsou přenášeny v pevně definovaných rámcích, které dovolují chránit předávaná data proti chybám. Struktura rámce často omezuje délku bloků dat síťové vrstvy - mluvíme o tzv. paketech. V případě poruchy vrstva zajišťuje opakované zaslání paketů.



Obrázek 2.1: Vrstvový OSI model.

2.1.3 Síťová vrstva

Síťová vrstva definuje jak se jednotlivé pakety pohybují v síti. Zatímco spojovací vrstva pracovala s fyzickými adresami, tak síťová vrstva již pracuje pouze s logickými adresami. Vrstva dále zajišťuje nadřazeným vrstvám nezávislost na směrování. Obsahuje funkce, které umožní překlenout rozdílné vlastnosti různých podsítí.

2.1.4 Transportní vrstva

Transportní vrstva se stává prostředníkem mezi třemi nadřazenými vrstvami, které se starají o zpracování dat a třemi podřazenými vrstvami, které se starají o správnou komunikaci. Všechny protokoly definované pro tuto vrstvu a vrstvy vyšší jsou koncové, tj. nezávisí na směrování. Vrstva dále zajišťuje rozkládání příliš dlouhých zpráv do paketů při vysílání a následně jejich spojení při příjmu. Mechanismy vrstvy se starají také o ochranu sítě proti nadměrné zátěži.

2.1.5 Relační vrstva

Relační vrstva doplňuje logické rozhraní pro aplikační programy o funkce, jakými jsou podpora poloduplexu a vkládání synchronizačních bodů. Zjednodušuje komunikaci programů i uživatelský pohled na komunikační kanál.

2.1.6 Prezentační vrstva

Prezentační vrstva transformuje přenášená data, tj. zajišťuje převody kódů a formátů dat pro nekompatibilní počítače, kompresi přenášených dat pro lepší lepší využití komunikačního kanálu a konečně i utajování přenášených dat.

2.1.7 Aplikační vrstva

Aplikační vrstva je vrstvou standardních aplikací a vrstvou rozhraní, která zjednodušují programování jednoúčelových aplikací.

Kapitola 3

Controller Area Network (CAN)

Materiály použité v této kapitole jsou čerpány z [3], [4] a [5].

3.1 Úvod

Controller Area Network (CAN) je sériový komunikační protokol, původně vyvinutý firmou Bosch pro nasazení v automobilech. Vzhledem k tomu, že přední výrobci integrovaných obvodů implementovali podporu protokolu CAN do svých produktů, je stále častěji využíván i v různých průmyslových aplikacích. Důvody jsou především nízká cena, dostupná potřebná součástková základna, snadné použití, spolehlivost, relativně velká přenosová rychlost a snadná rozšiřitelnost.

V současné době má protokol CAN pevné místo mezi ostatními průmyslovými komunikačními protokoly. Je definován standardem ISO 11898, který popisuje fyzickou a spojovací vrstvu protokolu specifikací CAN 2.0A. Později byla ještě vytvořena specifikace CAN 2.0B, která zavádí dva pojmy - standardní a rozšířený formát zprávy (lišící se délkou identifikátoru zprávy). Aplikační vrstva protokolu CAN je definována několika vzájemně nekompatibilními standardy (CAL/CANopen, DeviceNet aj.).

3.2 Základní vlastnosti protokolu CAN

Protokol CAN byl navržen tak, aby umožnil distribuované řízení systémů v reálném čase s přenosovou rychlostí do 1Mb/s a s vysokým stupněm zabezpečení přenosu proti poruchám. Protokol je typu *multimaster*, kde každé připojené zařízení na sběrnici může být řídicím *master* a řídit tak chování jiných. Není tedy nutné řídit celou síť z jednoho "nadřazeného" zařízení, což přináší zjednodušení řízení a zvětšuje spolehlivost (při poruše jednoho ze zařízení může zbytek sítě pracovat dál). Komunikace mezi zařízeními je typu *broadcast* (obr. 3.1), což znamená, že v danou chvíli vysílá pouze jedno zařízení a ostatní jsou příjemci zprávy. Vysílacím zařízením se stává to zařízení, které získá podle arbitračního algoritmu přístup k médiu.

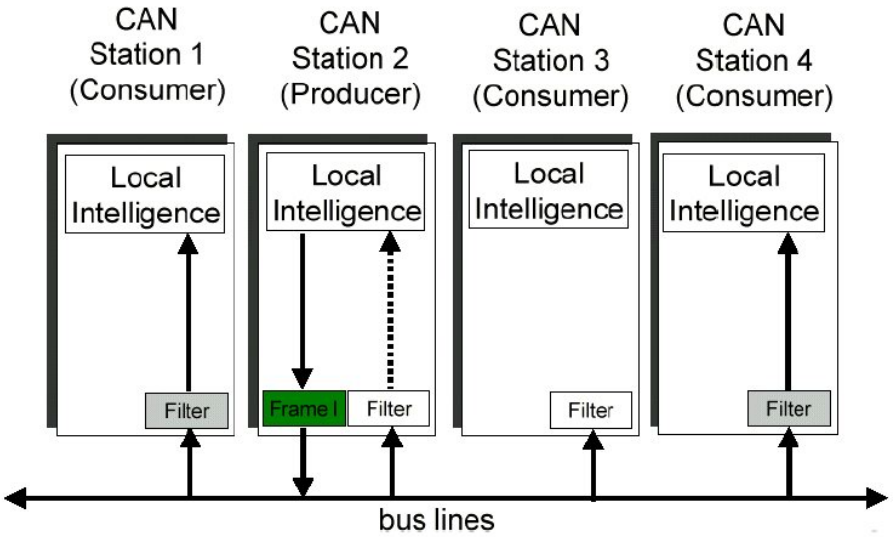
Pro řízení přístupu k médiu je použita sběrnice s náhodným přístupem, která řeší kolize na základě prioritního rozhodování. Komunikace mezi zařízeními na sběrnici probíhá pomocí dvou typů zpráv (datová zpráva a žádost o data). Řízení sítě (signalizace chyb, pozastavení komunikace) je zajištěno pomocí dvou speciálních zpráv (chybové zprávy a zprávy o přetížení).

Zpráva zasílaná určitým zařízením po sběrnici protokolem CAN neobsahuje žádnou informaci o cílovém zařízení, kterému je určena, a je přijímána všemi ostatními zařízeními

připojenými ke sběrnici. Každá zpráva je odvozena identifikátorem nesoucím informaci o významu přenášené zprávy a její prioritě. Nejvyšší prioritu má zpráva s identifikátorem 0. Protokol CAN zajišťuje, aby zpráva s vyšší prioritou byla v případě kolize dvou zpráv doručena přednostně. Dále je na základě indentifikátoru možné zajistit, aby určité zařízení přijímalo pouze ty zprávy, které se ho týkají (*acceptance filtering)*, na obrázku 3.1 jsou to zařízení s čísly 1 a 4. Ostatní zařízení přijímají všechny zprávy.

3.3 Fyzická vrstva

Základním požadavkem na fyzické přenosové médium protokolu CAN je, aby realizovalo funkci logického součinu. Standard protokolu CAN definuje dvě vzájemně komlementární hodnoty bitů na sběrnici - *dominant* a *recessive*. Jedná se v podstatě o jakýsi zobecnělý ekvivalent logických úrovní, jejichž hodnoty nejsou určeny a skutečná reprezentace záleží na konkrétní realizaci fyzické vrstvy. Pravidla pro stav na sběrnici jsou jednoduchá a jednoznačná. Vysílají-li všechna zařízení na sběrnici *recessive* bit, pak na sběrnici je úroveň *recessive*. Vysílá-li alespoň jedno zařízení *dominant* bit, je na sběrnici úroveň *dominant*. Příkladem může být optické vlákno, kde stavu *dominant* bude odpovídat stav *svítí* a stavu *recessive* stav *nesvítí*.

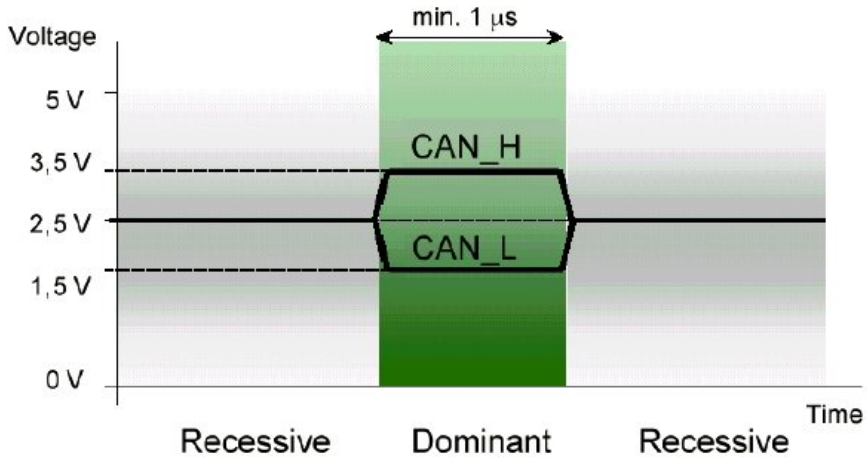


Obrázek 3.1: Komunikace typu *broadcast*.

3.3.1 Parametry sběrnice

Pro realizaci fyzického přenosového média se nejčastěji používá diferenciální sběrnice definovaná podle normy ISO 11898 [3]. Tato norma definuje jednak elektrické vlastnosti vysílacího budiče i přijímače, jednak principy časování, synchronizace a kódování jednotlivých bitů. Sběrnici tvoří dva vodiče (označované CAN_H a CAN_L), kde úroveň *dominant* nebo *recessive* na sběrnici je definovaná rozdílem napětí těchto dvou vodičů. Podle nominálních úrovní uvedených v normě je pro úroveň *recessive* velikost rozdílového napětí $V_{diff} = 0V$ a pro úroveň *dominant* $V_{diff} = 2V$ (obr. 3.2). Pro eliminaci odrazů na vedení je sběrnice na obou koncích přizpůsobena terminátory o hodnotě 120 Ω (obr. 3.4).

Vysoká přenosová rychlost vyžaduje velmi strmé náběžné hrany signálů. Těch lze dosáhnout pouze dostatečně výkonnými součástkami. V současnosti nabízí komponenty, které odpovídají normě ISO 11898 [3] několik výrobců (Bosh, Motorola, Philips, Texas Instrument aj.). Mezinárodní asociace uživatelů CiA [4] specifikovala také některé mechanické komponenty, jako jsou konektory a kabely. Nejčastěji se používají konektory D-SUB [4]. Veškeré elektrické charakteristiky sběrnice jsou detailně popsány v normě ISO 11898



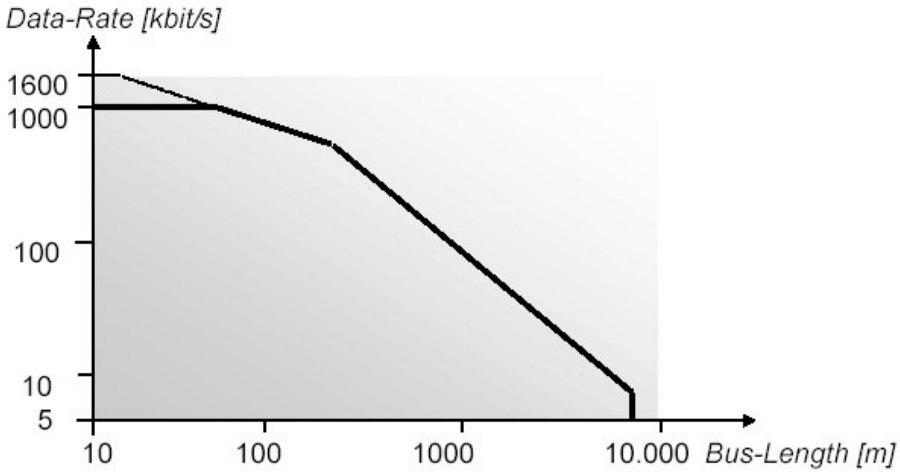
Obrázek 3.2: Napěťové úrovně odpovídající stavům *recessive* a *dominant*.

[3]. Základní parametry uvádí tabulka 3.1. Ke sběrnici může být teoreticky připojeno

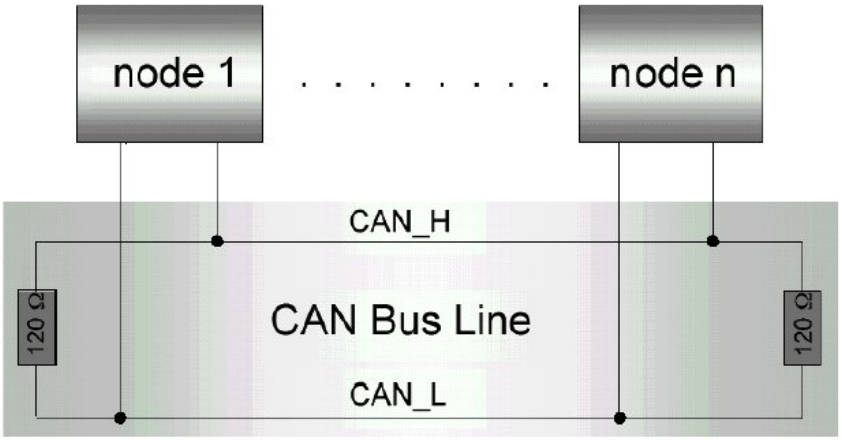
Přenosová rychlost	Délka sběrnice
1 Mb/s	40 m
500 kb/s	112 m
300 kb/s	200 m
100 kb/s	640 m
50 kb/s	1340 m
20 kb/s	2600 m
10 kb/s	5200 m
Maximální počet uzlů v síťi	30
Typická impedance vedení	120 Ω

Tabulka 3.1: Parametry sítě podle normy ISO 11898 [3].

neomezený počet zařízení, avšak s ohledem na zatížení sběrnice a zajištění správných statických i dynamických parametrů sběrnice omezuje norma počet zařízení na maximálně 30. Pro přenosovou rychlost 1 Mb/s je stanovena maximální délka sběrnice 40 m. Pro jiné přenosové rychlosti norma délku sběrnice neuvádí, avšak logickým úsudkem lze dojít k závěru, že pro delší sběrnice je nutná nižší přenosová rychlost. Maximální délky sběrnice pro jiné přenosové rychlosti než 1Mb/s, jak uvádí tabulka 3.1, jsou pouze informativní. Ve skutečnosti závisejí na mnoha parametrech (např. typ použitého kabelu).



Obrázek 3.3: Závislost přenosové rychlosti na délce sběrnice.



Obrázek 3.4: Uspořádání sběrnice CAN podle normy ISO 11898 [3].

3.4 Linková vrstva

3.4.1 Základní struktura

Linková vrstva protokolu CAN je tvořena dvěma podvrstvami označovanými jako MAC a LLC:

- MAC (*Medium Access Control*) zabezpečuje přístup k médiu s úkolem kódovat data, vkládat doplňkové bity do komunikace (*stuffing/destuffing*), řídit přístup všech zařízení k médiu s rozlišením priorit zpráv, detekce chyb a jejich hlášení a potvrzování správně přijatých zpráv.
- LLC (*Logical Link Control*) má za úkol filtrovat přijaté zprávy (*acceptance filtering*) a hlášení o přetížení (*overload notification*).

3.4.2 Řízení přístupu k médiu a řešení kolizí

Metoda přístupu na sběrnici je nedestruktivní typu `bitwise` tj. bit po bitu. Je-li sběrnice volná (stav *bus free*), může zahájit vysílání kterékoliv z připojených zařízení. Zahájí-li některé zařízení vysílání dříve než ostatní, získává sběrnici pro sebe a ostatní zařízení mohou začít vysílat až poté, co odešle kompletní zprávu. Jedinou výjimku zde tvoří chybové rámce, které může vyslat libovolné zařízení, detekuje-li chybu v právě přenášené zprávě.

Začne-li vysílat několik zařízení současně, je nutné zavést algoritmus pro získání média. Tento proces se nazývá *arbitrace*. Její princip spočívá v tom, že možnost vyslat zprávu na sběrnici má to zařízení, jehož zpráva má identifikátor s nejvyšší prioritou. Každý vysílač porovnává hodnotu právě vysílaného bitu s hodnotou na sběrnici a zjistí-li, že na sběrnici je jiná hodnota než je jeho vysílaný bit (jedinou možností je, že vysílač vysílá *rcessive* bit a na sběrnici je úroveň *dominant*), okamžitě přeruší další vysílání. Tímto způsobem je zajištěno, že zpráva s vyšší prioritou bude odeslána přednostně a nedojde k jejímu poškození, což by mělo za následek opakování zprávy a zbytečné prodloužení doby potřebné k přenosu zprávy. Zařízení, které při kolizi nezískalo přístup na sběrnici, může zprávu vyslat až poté co se sběrnice uvolní (*bus free*).

Obrázek 3.5: Postup při rozhodování o získaní sběrnice k vysílání (*arbitrační algoritmus*).

Obrázek 3.5 ukazuje příklad *arbitračního algoritmu*. V tomto případě začnou vysílat na sběrnici 3 zařízení (node 1, node 2, node 3), jejichž signály jsou znázorněny na prvních třech řádcích. Čtvrtý stav ukazuje skutečný stav sběrnice. Identifikátory vysílaných [1] se shodují až do bitu s číslem 5, kdy zařízení typu node 2 vysílá stav recessive a na sběrnici detekuje stav dominant. V tomto okamžiku node 2 přestává vysílat a stává se posluchačem sběrnice. Node 1 a node 3 stále pokračují ve vysílání až do bitu s číslem 2, který má node 1 nastaven na hodnotu recessive a také přichází o další možnost vysílání. V tuto chvíli už vysílá pouze node 3. Zařízení node 1 a node 2 mohou zopakovat pokus o získání sběrnice po odeslání celé zprávy z node 3. Během tohoto postupu tedy nedošlo k destruktivnímu přerušení vysílání, ale možnost vysílání ztratila pouze ta zařízení, jejichž identifikátor

[1] Každé zařízení může vysílat zprávy s různými identifikátory, a proto je nutné neslučovat identifikátor zprávy s identifikátorem zařízení. Tento fakt je řešen v konkrétní aplikační vrstvě

zprávy měl nižší prioritu (obsahoval na vyšších bitech úroveň recessive).

3.4.3 Zabezpečení přenášených dat

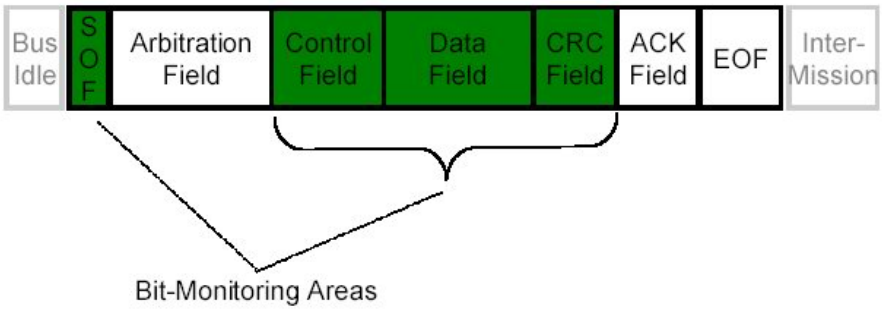
3.4.3.1 Typy zabezpečení

K zabezpečení kontroly dat a přenášených rámců slouží několik mechanismů, které fungují současně. Tím je zvýšena bezpečnost komunikace. Mezi tyto mechanismy patří:

- monitoring - Bit Error
- CRC kód
- vkládání bitů - Bit Stuffing
- kontrola zprávy
- potvrzení přijetí zprávy (ACK)

3.4.3.2 Monitoring - Bit Error

Při této kontrole porovnává vysílač hodnotu právě vysílaného bitu s úrovní detekovanou na sběrnici. Jsou-li obě hodnoty shodné, vysílač pokračuje ve vysílání. Pokud je na sběrnici detekována jiná úroveň, než odpovídá vyslané mu bitu a probíhá-li právě řízení přístupu na sběrnici (vysílá se *arbitration field*), vysílání se přeruší a přístup k médiu získá zařízení vysílající zprávu s vyšší prioritou. Pokud je neshoda vysílané a detekované úrovně zjištěna v jiném okamžiku, než při vysílání *arbitration field* a potvrzování přijetí zprávy (*ACK slot*), je vygenerována chyba bitu. Obrázek 3.6 znázorňuje oblast, ve které dochází



Obrázek 3.6: Kontrola monitorování bitů.

ke generování chybového hlášení v případě neshody bitu na sběrnici.

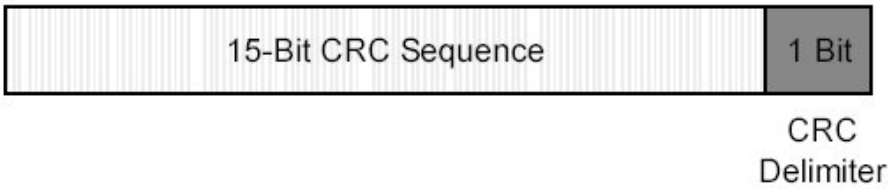
3.4.3.3 CRC kód

Princip zabezpečení dat metodou CRC *Cyclic Redundancy Check* spočívá v připojení 15 bitového slova na konec každého přenášeného rámce. Přesný tvar tohoto 15 bitového slova odpovídá zbytku po dělení příslušným polynomem, který je předem nadefinován a má tvar:

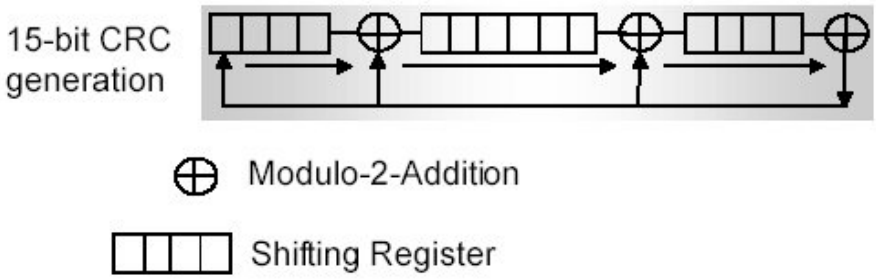
$$x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + 1. \tag{3.1}$$

přijímací zařízení kontroluje přijatou zprávu také pomocí dělení daným polynomem (3.1) a zbytek po dělení musí odpovídat 15 bitovému přijatému slovu CRC.

Teoreticky tato konstrukce CRC zaručuje Hammingovu vzdálenost 6, což znamená, že dojde k detekci pěti náhodně po sobě jdoucích chyb. Obrázek 3.8 ukazuje hardwarovou realizaci CRC. Zde je použito dělení modulo 2 pomocí posuvného registru. Pokud zařízení

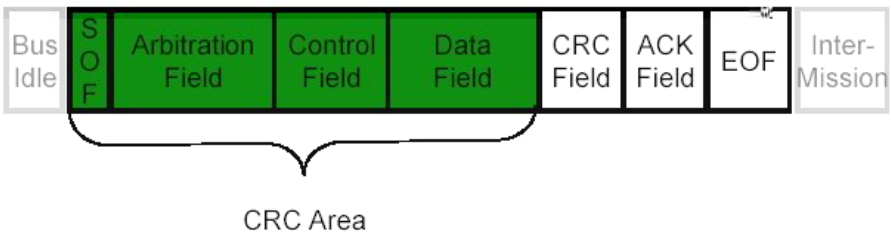


Obrázek 3.7: CRC field



Obrázek 3.8: Generování CRC kódu.

detekuje chybu při kontrole pomocí CRC, je generováno hlášení o chybě. Oblast, ve které dochází ke kontrole pomocí CRC algoritmu, ukazuje obrázek 3.9. Tato část tvoří základ pro dělení známým dělitelem.



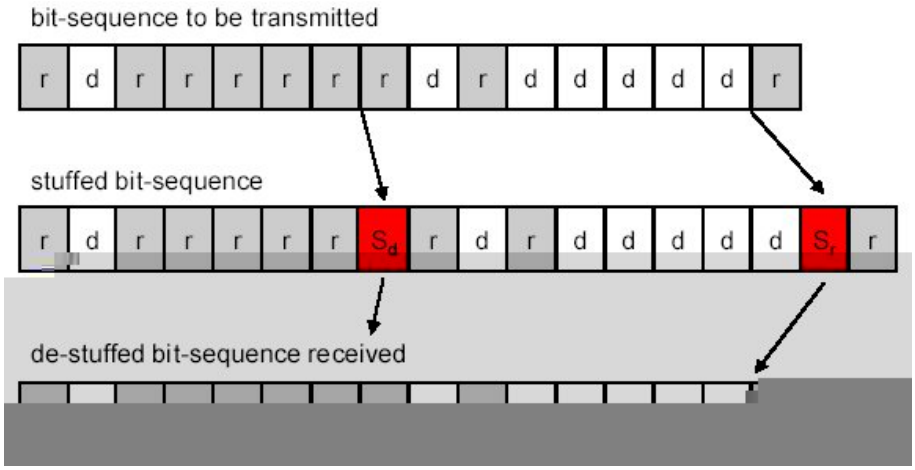
Obrázek 3.9: Oblast CRC chybové kontroly.

3.4.3.4 Vkládání bitů - Bit Stu□ng

Signál na sběrnici je typu NRZ (Non Return to Zero - logické úrovně 0 a 1 jsou reprezentovány napěťovými úrovněmi a nedochází během platnosti bitu ke změnám hran - viz obr. 3.10). Tato skutečnost způsobuje problém při synchronizaci. Pro jeho odstranění se používá tzv. *vkládání bitů - bit stuffing*. To je založeno na vložení jednoho bitu opačné

Obrázek 3.10: NRZ kódování.

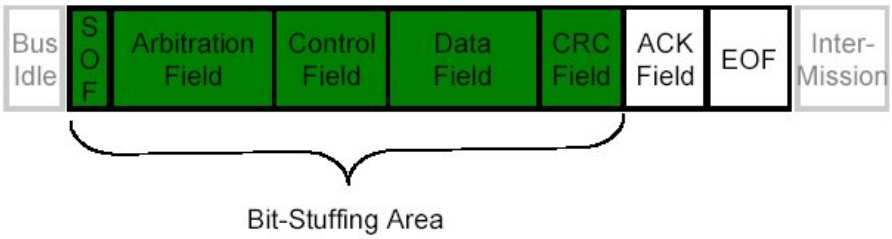
úrovně po vyslání pěti po sobě jdoucích bitech stejné úrovně. Toto opatření slouží k detekci chyb, ale také ke správné časové synchronizaci přijímačů jednotlivých uzlů. Je-li detekována chyba vkládání bitů, je generováno hlášení o chybě vkládání bitů.



Obrázek 3.11: Pravidla pro vkládání bitů (*bit stuffing*).

Odstraňování vloženého bitu provádí přijímač, který monitoruje příchozí bity a po každých pěti bitech stejné úrovně automaticky odstraní šestý bit v pořadí. Obrázek 3.11 ukazuje vkládání a odstraňování bitů (*bit-stuffing/de-stuffing*), vkládaný bit je označen S_d a odstraňovaný bit S_r.

Oblast, ve které probíhá vkládání bitů je znázorněna na obrázku 3.12.



Obrázek 3.12: Pravidla pro vkládání bitů (*bit stuffing*).

3.4.3.5 Kontrola zprávy

Tato kontrola provádí porovnávání zprávy s formátem zprávy uvedeným ve specifikaci [4]. Je-li na některé bitové pozici zprávy detekována nepovolená hodnota, je vygenerováno hlášení chyby rámce (chyba formátu zprávy).

3.4.3.6 Potvrzení přijetí zprávy (ACK)

Dojde-li ke správnému přijetí zprávy libovolným zařízením, je tento fakt potvrzen změnou jednoho bitu zprávy ACK. Vysílač vždy na tomto bitu vysílá úroveň *recessive* a detekuje-li úroveň *dominant*, pak je vše v pořádku. Přijetí zprávy potvrzují všechna zařízení připojená ke sběrnici, ať již mají filtrování zpráv (*acceptance filtering*) zapnuté nebo vypnuté.

Obrázek 3.13: ACK bit v CAN rámci.

3.4.4 Signalizace chyb

Každé přijímací zařízení vyšle po zjištění chybového příjmu zprávu Error Frame, která poruší správnost vysílaných dat a způsobí, že chybnou zprávu dostanou i ostatní uzly. Každé zařízení má zabudované dva čítače chyb detekovaných při příjmu a při vysílání zpráv. V závislosti na počtu chybových zpráv (tj. na stavu čítačů), které dané zařízení vyslalo nebo přijalo, se může nacházet mezi třemi stavy: aktivní, pasivní a odpojeno. Jestliže zařízení generuje příliš mnoho chyb, je automaticky odpojeno (přepnuto do stavu *bus-off*).

Každý CAN řadič se tedy může nacházet v jednom ze tří stavů.

- **Aktivní** *(error active)* - zařízení v tomto stavu se mohou aktivně podílet na komunikaci po sběrnici a v případě, dojde-li k detekci libovolné chyby v právě přenášené zprávě (chyba bitu, CRC, vkládání bitů, rámce), vysílají na sběrnici aktivní příznak chyby (*active error flag*). Tento příznak chyby je tvořen šesti po sobě jdoucími bity *dominant*, čímž dojde k poškození přenášené zprávy (poruší se pravidlo vkládání bitů).

- **Pasivní** *(error passive)* - nachází-li se zařízení v tomto stavu, pak se také podílejí na komunikaci, ale z hlediska hlášení chyb vysílají pouze pasivní příznak chyby *(passive error flag)* tvořený šesti po sobě jdoucími bity *recessive*, čímž se zamezí destrukci právě vysílané zprávy.

- **Odpojená** *(bus-off)* - takováto zařízení nemají žádný vliv na sběrnici a jejich výstupní budiče jsou vypnuty.

Přechody mezi jednotlivými stavy jsou dány hodnotami čítačů chyb. Jak bylo již dříve zmíněno existují dva čítače chyb - čítač chyb při vysílání (TEC) a při příjmu zpráv (REC). Pravidla pro přechod zařízení mezi jednotlivými stavy (viz obr. 3.14) na základě hodnot čítačů definuje norma ISO 11898 [3].

Obrázek 3.14: Stavový diagram zařízení z hlediska hlášení chyb

Tabulka 3.2 ukazuje hodnoty čítačů chyb, které odpovídají jednotlivým stavům.

Stav zařízení	Hodnota čítače chyb při vysílání	Hodnota čítače chyb při příjmu
aktivní	< 127	< 127
pasivní	$>= 127$	$>= 127$
odpojeno	$>= 256$	-

Tabulka 3.2: Možné hodnoty čítačů pro přechod mezi stavy.

Hodnoty čítačů (TEC, REC) jsou inkrementovány v případě vyslání aktivního chybového rámce (u čítače TEC) nebo je-li detekována chyba při příjmu a jsou splněny podmínky pro inkrementaci čítače [2] (u čítače REC). Překročí-li hodnoty čítačů (TEC, REC) hranici 96, tak zařízení zůstávají ve stavu *Error Active*, ale je detekováno silné přetížení sítě. Pokud hodnota čítače TEC překročí hranici 256, tak je zařízení odpojeno (stav *bus-off*). Pokud zařízení přijme jednu zprávu bez jakýchkoli problémů, je hodnota čítače REC snížena o 1 (nebyla-li 0) a pokud se stav čítače nacházel v intervalu $1 \div 127$. Podobně při úspěšném odeslání zprávy dojde ke snížení čítače TEC o 1 (nebyla-li 0).

3.5 Základní typy zpráv

3.5.1 Rozdělení zpráv

Specifikace protokolu CAN definuje celkem čtyři typy zpráv:

- datová zpráva
- vyžádání dat

[2] Existují výjimky, kdy k inkrementaci čítače REC nedochází - detekce *Bit Error* při odesílání *Active Error Flag*

- zpráva o chybě
- zpráva o přetížení

První dva typy zpráv se týkají přenosu dat po sběrnici, kde *datová zpráva* představuje základní prvek komunikace zařízení po sběrnici. Umožňuje vyslat na sběrnici až 8 byte dat. Při jednoduchých příkazech zařízením (typu vypni/zapni apod.) není nutné přenášet žádné byty dat (příkaz je zakódován v identifikátoru zprávy), což zkracuje dobu potřebnou k přenosu zprávy a zároveň zvětšuje propustnost sběrnice, zvláště při velkém zatížení.

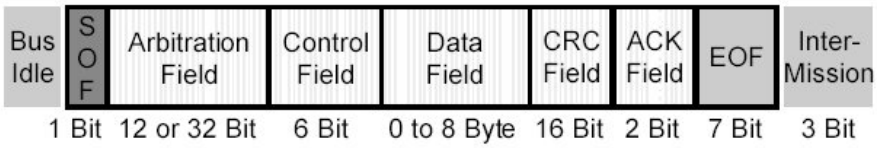
Zprávou, typu *vyžádání dat*, žádá některé zařízení ostatní účastníky na sběrnici o zaslání dat. Identifikátor určuje číslo uzlu, který je žádán o data. Pokud v jednom okamžiku začne jedno zařízení vysílat zprávu žádost o data a současně druhé začne vysílat data (identifikátor zprávy je shodný), začne se uplatňovat algoritmus pro rozhodování o přístupu ke sběrnici. V tomto okamžiku získá možnost vysílat to zařízení, které vysílá data a odpadá žádost o data. Tato skutečnost nastane z důvodu hodnoty RTR, jehož hodnota je u přenosu dat *dominant* a u přenosu žádosti o data je *recessive*.

Druhé dva typy zpráv slouží k řízení komunikace po sběrnici, konkrétně k signalizaci detekovaných chyb, eliminaci chybných zpráv a vyžádání prodlevy v komunikaci.

3.5.2 Datová zpráva (*data frame*)

3.5.2.1 Obecný tvar datové zprávy

Norma [4] definuje přesný tvar každé zprávy protokolu CAN. Na obrázku 3.15 je uvedena obecná struktura datové zprávy.

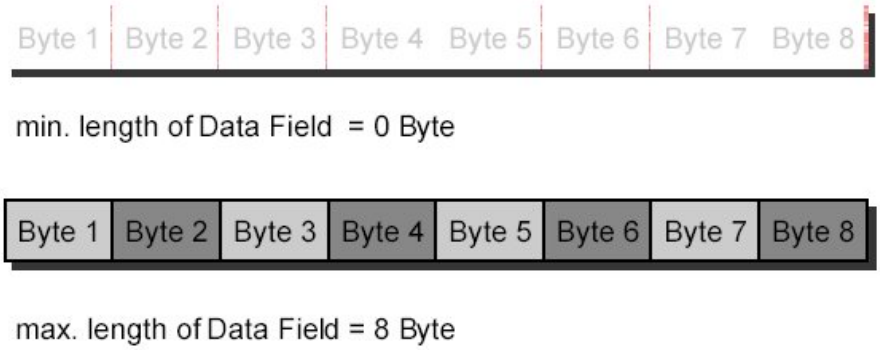


Obrázek 3.15: Tvar datového rámce protokolu CAN.

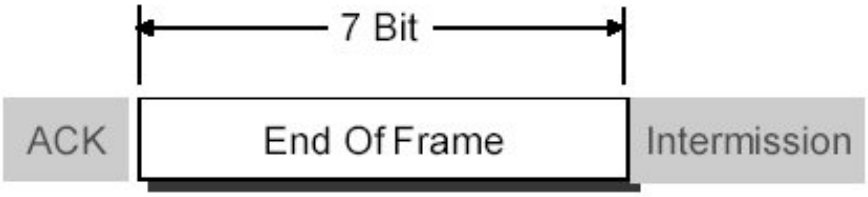
CAN rámec obsahuje tyto části:

- **Začátek zprávy (SOF** - *Start Of Frame*) - začátek každé zprávy, 1 bit *dominant*
- **Arbitrační část** (*arbitration field*) - v této části se určuje priorita zprávy
- **Kontrolní část** (*control field*) - tato část CAN rámce obsahuje 6 bitů, které lze rozdělit na:
  - IDE (*Identifier Extension*) - informuje o tom, zda přenášená zpráva je typu standard CAN 2.0A nebo CAN 2.0B
  - R0 - rezervovaný bit
  - DLC (*Data Length*) - 4 bity určující délku přenášených dat, číslo udává kolik byte je umístěno v datové oblasti
- **Datová část** (*data field*) - datový obsah zprávy (max. 8 byte), viz obrázek 3.16
- **CRC** (*CRC field*) - kontrola přenášené zprávy

- **ACK field** - potvrzení přijetí zprávy
- **Konec zprávy** (*End of Frame*) - 7 bitů *recessive*, viz obrázek 3.17



Obrázek 3.16: Datová část CAN rámce.

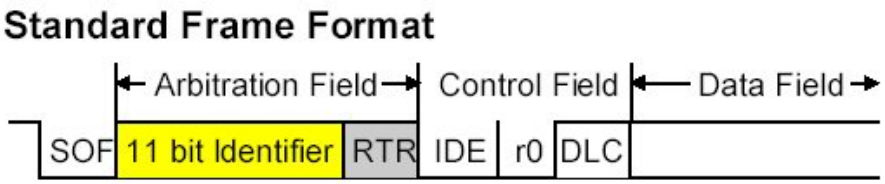


Obrázek 3.17: Konec CAN rámce.

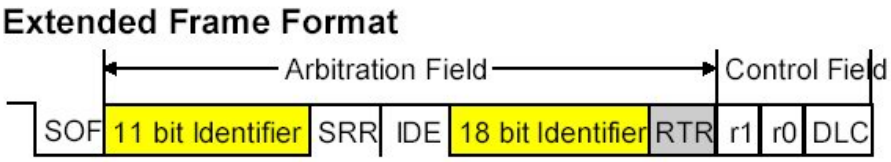
Mezi každou novou přenášenou zprávou musí být minimálně 3 bity mezera.

3.5.2.2 Standardní formát zprávy a rozšířený formát zprávy

Protokol CAN rozlišuje dva typy datových zpráv. První je označován jako standardní formát zprávy (*standard frame*) a definován ve specifikaci CAN 2.0A (viz obrázek 3.18) a druhý je označován jako rozšířený formát zprávy (*extended frame*) a je definován ve specifikaci CAN 2.0B (viz obrázek 3.19). Tyto dva formáty datových zpráv se od sebe odlišují v délce identifikátoru zprávy, který je u standardního 11 bitů a u rozšířeného formátu 29 bitů. Je-li použitým řadičem podporován protokol CAN 2.0B, pak mohou být oba typy zpráv použity současně na jedné sběrnici. Ve specifikaci CAN 2.0A se bit R1 používá pro rozlišení zda se jedná o standardní (*úroveň dominant*) nebo rozšířený (*úroveň recessive*) rámec. Ve specifikaci CAN 2.0B se tento bit označuje IDE - (*Identifier Extended*). Jak již bylo řečeno rozšířený formát používá identifikátor zprávy o celkové délce 29 bitů, rozdělený do dvou částí o délkách 11 (stejný identifikátor je použit ve standardním formátu) a 18 bitů. Bit RTR je zde nahrazen bitem SRR (*Substitute Remote Request*), který je v rozšířeném formátu vždy *recessive*. To zajišťuje, aby při vzájemné kolizi standardního a rozšířeného formátu zprávy na jedné sběrnici, získal přednost standardní formát. Bit IDE (*Identifier Extended*) je vždy *recessive* a bit RTR je přesunut za konec druhé části identifikátoru. Pro řízení přístupu k médiu jsou použity ID (11 bitů), SRR, IDE, ID (18 bitů), RTR. V tomto pořadí je určena priorita datové zprávy.



Obrázek 3.18: Standardní formát datové zprávy.



Obrázek 3.19: Rozšířený formát datové zprávy.

3.5.3 Žádost o data (*remote frame*)

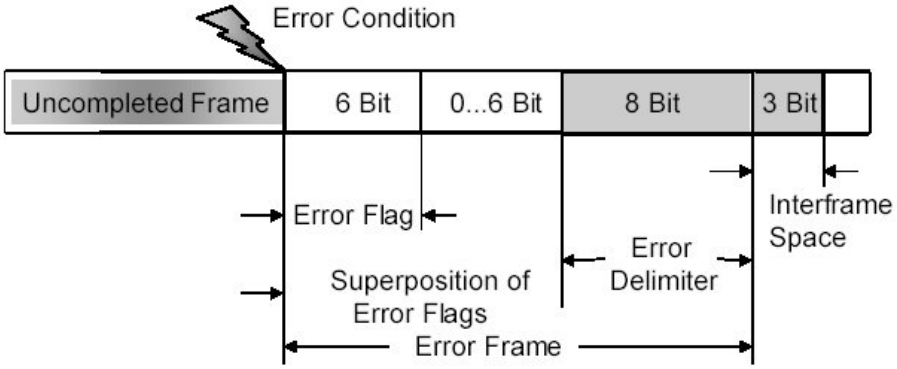
Používá se v případě, že jedno zařízení žádá o data druhé zařízení. Formát zprávy je podobný jako u datové zprávy, s tím že neobsahuje datovou část (viz obrázek 3.20. Bit RTR (pole řízení přístupu na sběrnici) je zde nastaven jako *recessive*, aby při shodném počátku vysílání datové zprávy a žádosti o data dostal přenos dat přednost před žádostí. Datová zpráva má bit RTR vždy nastaven na úroveň *dominant*.

Obrázek 3.20: Formát zprávy žádost o data.

3.5.4 Zpráva o chybě (*error frame*)

Zpráva o chybě slouží k signalizaci chyb na sběrnici CAN. Zjistí-li nějaké zařízení na sběrnici v přenášené zprávě chybu (chyba bitu, chyba CRC, chyba vkládání bitů, chyba rámce), ihned vygeneruje na sběrnici zprávu o chybě. Podle stavu v jakém se zařízení, které zjistilo chybu právě nachází, vygeneruje na sběrnici buď aktivní *6 bitů dominant* nebo pasivní *6 bitů recessive* příznak chyby.Při generování aktivního příznaku chyby poškozena právě přenášená zpráva (porušení pravidla o vkládání bitů) a zprávy o chybě začnou vysílat i ostatní zařízení. Hlášení chyb je poté indikováno superpozicí všech chybových příznaků vysílaných jednotlivými zařízeními. Délka tohoto úseku může být mezi 6 až 12 bity (viz obrázek 3.21).

Pro vysílání chybového příznaku vysílá každé zařízení na sběrnici bity *recessive*. Zjistí-li zařízení, že je na sběrnici také úroveň *recessive*, vyšle dalších 7 bitů *recessive*, které slouží

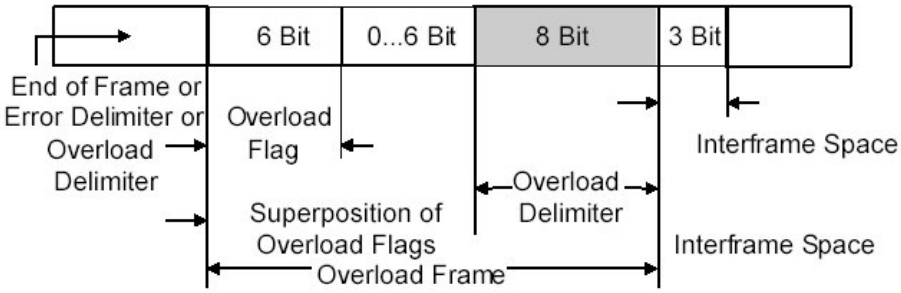


Obrázek 3.21: Aktivní rámec zprávy o chybě.

jako oddělovače chyb (ukončení zprávy o chybě).

3.5.5 Zpráva o přetížení (*overload frame*)

Zpráva o přetížení slouží k oddálení další datové zprávy nebo žádosti o data. Tento způsob zpravidla využívají zařízení, která nejsou schopna díky svému vytížení přijímat a zpracovávat další zprávy. Struktura zprávy je podobná zprávě o chybě. Její vysílání může být zahájeno po konci zprávy (*end of frame*), oddělovače chyb nebo předchozího oddělovače zpráv o přetížení (viz obrázek 3.22).



Obrázek 3.22: Rámec zprávy o přetížení.

Zpráva o přetížení se skládá z příznaku přetížení (6 bitů *dominant*) a případné superpozice všech příznaků přetížení, pokud jsou generovány několika zařízeními současně. Za příznaky přetížení následuje dalších 7 bitů *recessive* tvořících oddělovač zprávy o přetížení. Každé zařízení může vysílat maximálně dva po sobě jdoucí rámce typu zpráva o přetížení.

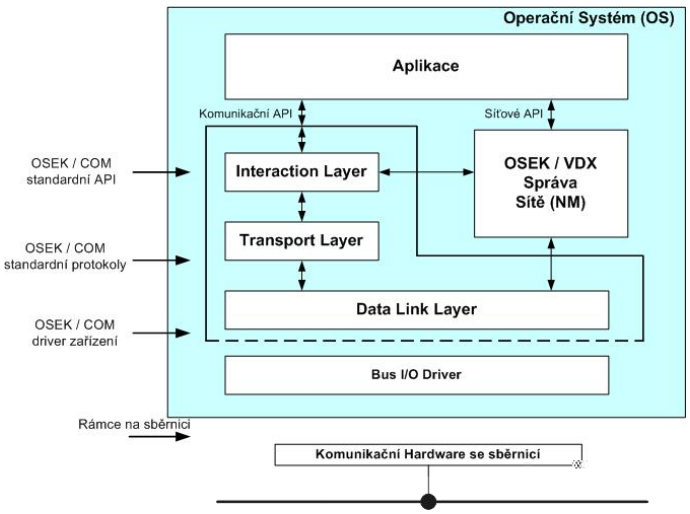
Kapitola 4

Real-Time operační systém OSEK/VDX

4.1 Úvod

Jedná se o Real-Time operační systém vyvinutý speciálně pro distribuované řízení v automobilech. Vznikl sjednocením projektů francouzských a německých automobilových společností. Jeho otevřená architektura poskytuje rozhraní nezávislé na typu sítě (viz obrázek 4.1). Architektura OSEK/VDX zahrnuje tyto oblasti (viz obrázek 4.1):

- Komunikace (*Communication*) - výměna dat uvnitř a mezi řídícími jednotkami.
- Správa sítě (*Network Management*) - rozhodování a monitorování sítě.
- Operační systém (*Operating system*) - Real-time exekutiva pro ECU (Electronic Control Unit).
- OSEK/VDX implementační jazyk (*OIL*) - jazyk, kterým se vytváří definice vlastní aplikace.



Obrázek 4.1: Vrstvový model OSEK/VDX.

V současné době dochází k růstu distribuovaného řízení v automobilové elektronice. To umožňuje použití různých řídících jednotek (ECU), inteligentních senzorů nebo akčních členů, které jsou mezi sebou spojeny přes odpovídající síťové rozhraní. Takovéto spojení do jedné sítě tvoří základ top-down řízení, které poskytuje vyšší výkon, více komfortu, vyšší stupeň bezpečnosti a nižší emise, které jsou požadovány v dnešních automobilech. Tyto aplikace jsou nezávislé na použitých platformách jednotlivých ECU.

V distribučních architekturách automobilu můžeme identifikovat nejméně tři aplikačně nezávislé úlohy. První z nich je operační systém, který poskytuje prostředí pro spouštění funkcí nebo tasků. Druhou úlohou je komunikační software pro výměnu dat mezi řídícími jednotkami. Třetí úlohou je správa sítě, která má zajistit bezpečnost operací pomocí monitorování konfigurace. Standardizace rozhraní a protokolů těchto nezávislých úloh (operační systém, komunikace a správa sítě) přispívá k podpoře přenositelnosti aplikace.

Hlavní výhody dané standardizací:

- snížení nákladů na vývoj nové aplikace,
- zvýšení kvality softwaru,
- použitelnost sofwarových modulů od různých dodavatelů v jednom mikrokontroleru.

4.2 Operační systém (*Operating system*)

Materiály použité v této kapitole jsou čerpány z [6] (OS22.pdf).

4.2.1 Filozofie systému

Automobilové aplikace jsou charakteristické svými požadavky na spouštění v reálném čase. Operační systém OSEK proto poskytuje nezbytnou funkční podporu pro události řízenými řídícími systémy. Služby operačního systému představují základ pro integraci softwarových modulů vytvořených různými výrobci. Aby mohl reagovat na specifické vlastnosti jednotlivých řídících jednotek, které jsou dané jejich výkonem a požadavkem na minimální spotřebu zdrojů, tak se nezaměřuje hlavně na 100% kompatibilitu mezi aplikačními moduly, ale zaměřuje se na jejich přímou přenositelnost.

Protože je operační systém OSEK určen pro jakékoliv řídící jednotky, musí být schopen podporovat časově kritické aplikace na širokém spektru hardwaru. Vysoký stupeň modularity a schopnost pro flexibilní uspořádání jsou nezbytnými předpoklady pro vytvoření operačního systému, který je vhodný pro low-endové mikroprocesory a komplexní řídící jednotky. Tyto požadavky jsou podporované pomocí tzv. *"conformance classes"* (tříd konformity) a určitými schopnostmi pro přizpůsobení se požadavkům aplikace.

Z důvodů časově kritických aplikací je dynamické generování systémových objektů vynecháno. Z tohoto důvodu se všechny systémové objekty vytvářejí ve fázi generování systému. Chybová hlášení uvnitř operačního systému jsou do značné míry odstraněna, aby zbytečně neovlivňovala rychlost celého systému. Na druhé straně, byla definovaná systémová verze s rozšířenými chybovými hlášeními, která je určena pro testovací fázi vývoje aplikace a pro méně časově kritické aplikace.

4.2.1.1 Standardizované rozhraní

Rozhraní mezi aplikačním softwarem a operačním systémem je definované pomocí systémových služeb. Toto rozhraní je stejné pro všechny implementace operačního systému na různých typech procesorů. Systémové služby mají syntaxi ISO/ANSI C.

4.2.1.2 Rozšiřitelnost

Různé *"conformance classes"*, různé plánovací mechanismy a konfigurační vlastnosti dělají operační systém OSEK použitelným pro mnoho druhů aplikací na širokém spektru hardwaru. Operační systém OSEK je navržený tak, aby vyžadoval minimum hardwarových zdrojů (RAM, ROM, CPU time), a proto je použitelný i na 8 bitových procesorech, jakými mohou být například procesory Motorola řady HC08.

4.2.1.3 Kontrola chyb

Operační systém OSEK nabízí dvě úrovně kontroly chyb. Jedná se o tzv. rozšířenou kontrolu chyb, která se používá ve vývojové fázi a standardní kontrolu chyb používanou ve výrobní fázi. Rozšířená kontrola chyb využívá při kontrole většího počtu volání systémových služeb, tudíž požaduje vyšší nároky na paměťový prostor a tím i vyšší časovou náročnost něž standardní kontrola chyb. Potom co byly všechny chyby eliminovány, systém může být překompilován se standardní kontrolou.

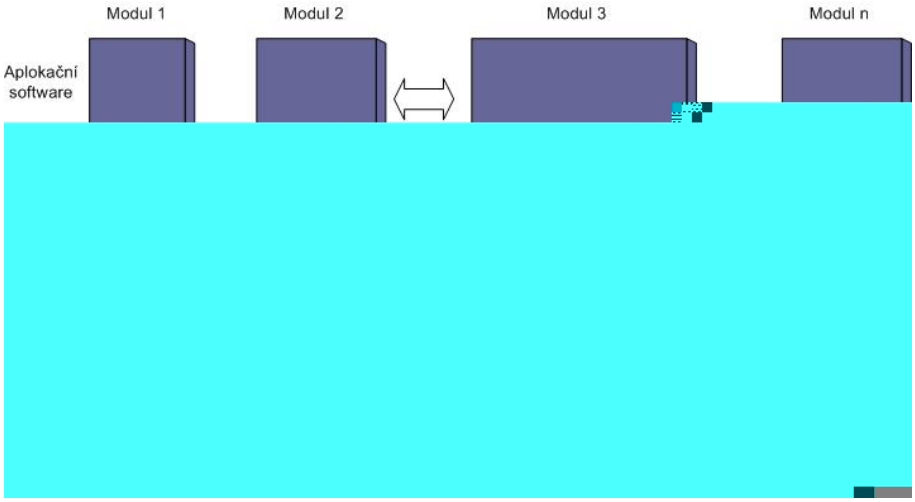
4.2.1.4 Přenositelnost aplikace

Jedním z hlavních cílů OSEKu je podpora přenositelnosti aplikace. Z tohoto důvodu je zavedeno standardizované rozhraní mezí aplikací a operačním systémem pomocí volání systémových služeb, tím zároveň dochází ke snížení nákladů na vývoj. Přenositelností se rozumí schopnost přenášet aplikační program mezi různými ECU (Electronic Control Unit) bez větších úprav uvnitř aplikace. Standardním rozhraním jako jsou, volání služeb, typy definic a konstant, se zajistí přenosnost na úrovni zdrojového kódu aplikace.

Aplikační software je umístěn nad operačním systémem a paralelně se systémem vstupů/výstupů, který není standardizován ve specifikaci OSEKu. Aplikační software může mít několik rozhraní, tím se rozumí rozhraní s operačním systémem pro řízení v reálném čase a správa zdrojů, rozhraní s ostatními sowtwarovými moduly zajišťující potřebné funkce v systému a v neposlední řadě také s hardwarem pokud aplikace pracuje přímo s moduly procesoru.

Pro zajištění lepší přenositelnosti aplikačního softwaru definuje OSEK jazyk, ve kterém se jednotlivé aplikace definují pomocí systémových objektů (např. úlohy, alarmy, zdroje, zprávy a další). Tím se zajistí standardizované rozhraní pro všechny aplikace a tím i jejich přenositelnost. Tento jazyk se nazývá OIL (*OSEK Implementation Language*).

Během procesu přenášení aplikace mezi jednotlivými ECU je nutné brát v úvahu vlastnosti vývojového prostředí, vývojové procesy aplikace a architekturu hardwaru jednotlivých ECU. To znamená, že OSEK specifikace není dostatečná k tomu, aby popisovala kompletně OSEK implementaci. Implementace musí doplněna příslušnou dokumentací.



Obrázek 4.2: Možné rozhraní v ECU.

4.2.1.5 Speciální podpora pro požadavky automobilového průmyslu

Specifické požadavky na operační systém OSEK se objevují v souvislosti s vývojem softwaru pro řídící jednotky v automobilu. Požadavky na spolehlivost, schopnost běhu v reálném čase popisují následující vlastnosti:

- Operační systém OSEK je konfigurován staticky, tzn. že uživatel staticky definuje počet úloh, alarmů, zdrojů a požadovaných služeb.
- Specifikace operačního systému OSEK podporuje implementaci schopnou běžet v paměti ROM, tzn. kód může být spouštěn z této paměti.
- Operační systém OSEK podporuje přenositelnost aplikačních úloh.
- Specifikace operačního systému OSEK poskytuje předvídatelné a dokumentované chování pro možné implementace operačního systému, se kterým se setkáme při řízení v reálném čase v automobilech.
- Specifikace operačního systému OSEK dovoluje implementace s ohledem na předpokládané výkonové parametry.

4.2.2 Architektura operačního systému OSEK

4.2.2.1 Procesní úrovně

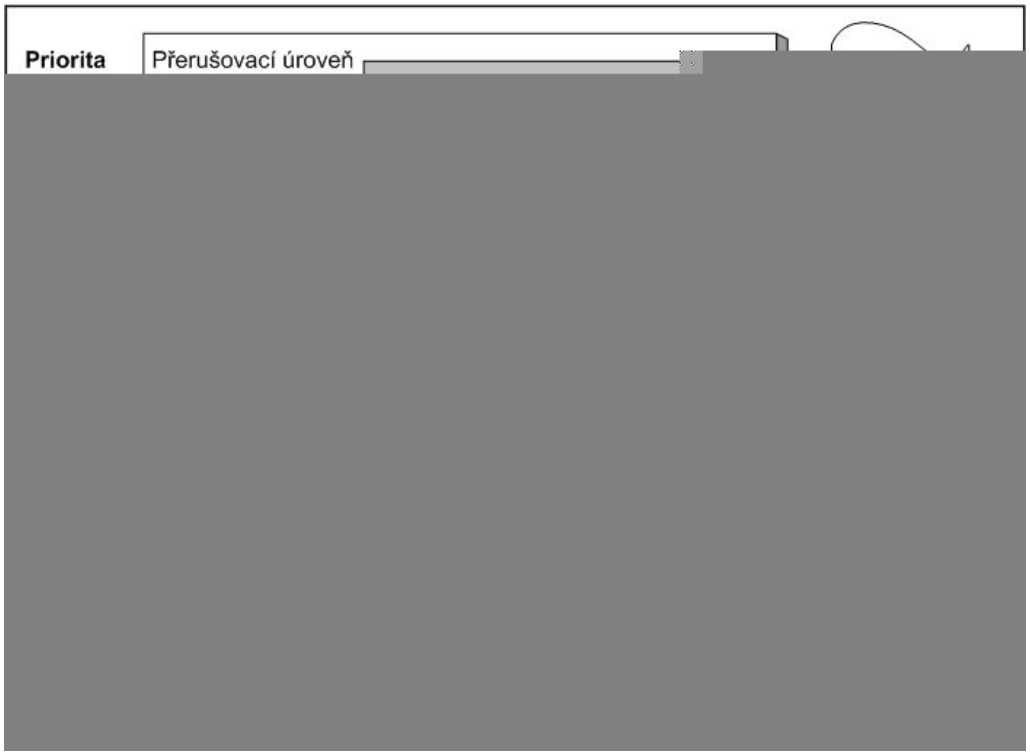
Operační systém OSEK slouží jako základ pro aplikační programy, které jsou vzájemně nezávislé. OSEK umožňuje spouštění procesů v reálném čase, které je vnímáno jako kdyby se tyto procesy vykonávaly paralelně. Dále poskytuje soubor definovaných rozhraní pro uživatele. Tato rozhraní jsou používána entitami, které soupeří o procesor. Existují dva typy entit:

- Interrupt Service Routines (ISR) řízené operačním systémem
- Úlohy (základní a rozšířené)

Hardwarové zdroje řídicí jednotky mohou být spravovány službami operačního systému. Tyto služby jsou volány buď pomocí aplikačního programu nebo vnitřně uvnitř operačního systému. OSEK definuje tři procesní úrovně (viz obrázek 4.3):

- Interrupt level (úroveň přerušení)
- úroveň plánovače
- úroveň úloh

Úroveň úloh definuje způsoby plánování (preemtivní, nepreemtivní a smíšené), podle uživatelem definované priority.



Obrázek 4.3: Úrovně zpracování operačního systému OSEK/VDX.

Jsou stanovena následující pravidla, kdo má jakou prioritu:

- Přerušení mají přednost před úlohami.
- Úroveň přerušení se skládá z jedné nebo více úrovní priority přerušení.
- Služby přerušení mají staticky přidělenou úroveň priority přerušení.
- Přidělení rutiny obsluhy přerušení úrovním priority přerušení je závislé na implementaci a architektuře hardwaru.
- Pro úlohy a zdroje řízené prioritou platí, že vyšší hodnota znamená vyšší prioritu.
- Priorita úlohy je staticky přidělena uživatelem.

Procesní úrovně jsou definované pro manipulaci s úlohami a rutin obsluh přerušení jako posloupnost nepřetržitých hodnot. Operační systém poskytuje služby a dohlíží na plnění výše uvedených prioritních pravidel.

4.2.3 Třídy konformity (*Conformance classes*)

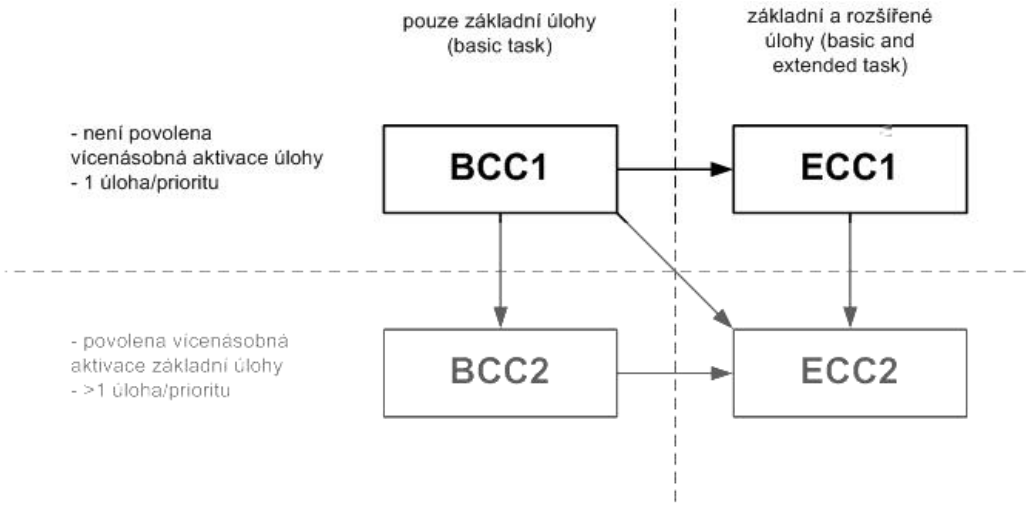
Různé požadavky aplikačního programu na systém a různé schopnosti daného systému (např. procesor, paměť) požadují různé vlastnosti operačního systému. V operačním systému OSEK se tyto vlastnosti popisují jako tzv. "třídy konformity" (Conformance Classes). Třídy konformity mají plnit tyto cíle:

- Poskytovat vhodné vlastnosti operačního systému pro snadnější porozumění a diskusi o OSEK operačním systému.
- Dovolují částečnou implementaci pomocí předdefinovaných struktur.
- Vytvoření cesty pro aktualizaci aplikačního programu z třídy s nižší funkčností do třídy s vyšší funkčností bez zásahu do stávajících částí aplikace.

Jsou-li definované všechny třídy konformity, pak je operační systém definován podle specifikace normy OSEK. Třídy konformity se nemohou v průběhu vykonávání aplikace měnit. Souhlasné třídy jsou určeny těmito atributy:

- Vícenásobné požadavky na aktivaci úloh.
- Typy úloh.
- Počet úloh na jednu prioritu.

Všechny další vlastnosti jsou povinné, není-li explicitně uvedeno jinak.



Obrázek 4.4: Kompatibility tříd konformity.

OSEK definuje následují třídy konformity:

- BCC1: tato třída obsahuje pouze základní úlohy, omezení jedné žádosti o aktivaci úlohy na jednu úlohu a každá úloha smí mít definovanou pouze jednu prioritu, takže každá úloha má jinou prioritu.
- BCC2: stejná jako BCC1, navíc ale dovoluje více jak jednu úlohu na prioritu a vícenásobnou žádost o aktivaci úlohy.
- ECC1: stejná jako BCC1, ale navíc může obsahovat rozšířené úlohy.
- ECC2: stejná jako ECC2, navíc ale dovoluje více než jednu úlohu na prioritu a požadavek na vícenásobnou aktivaci úloh pro základní úlohy.

Přenositelnost aplikace může být zajištěna, pokud nejsou překročeny minimální požadavky pro třídy konformity, které ukazuje následující tabulka.

	BCC1	BCC2	ECC1	ECC2
Vícenásobný požadavek na aktivaci úlohy	ne	ano	BT: ne ET:ne	BT: ano ET:ne
Počet úloh, které nejsou v *susspended* stavu	8		16	

Obrázek 4.5: Minimální požadavky pro třídy konformity.

4.2.4 Správa úloh (*task management*)

Komplexní řídící software může být vhodně rozdělen na části, které se provádějí podle požadavků v reálném čase. Tyto části mohou být realizovány prostřednictvím úloh. Úloha poskytuje strukturu pro spouštění funkcí. Operační systém zajišťuje souběžné a asynchronní provádění úloh. Plánovač poté organizuje pořadí spouštění úloh.

Operační systém OSEK poskytuje mechanismus pro přepínání úloh, tzv. plánovač (*scheduler*) zahrnující mechanismus, který je aktivní, když není aktivní žádný jiný systém nebo aplikační funkce. Tento mechanismus se nazývá pasivní mechanismus. Operační systém OSEK poskytuje dvě koncepce úloh:

- **Základní úlohy** (*basic tasks*): takovýto typ úloh uvolní procesor pouze, jestliže:
  - jsou ukončeny
  - OSEK operační systém přepne na úlohu s vyšší prioritou nebo
  - nastane přerušení, což způsobí,že se dojde k přepnutí procesoru do rutin na obsluhu přerušení (ISR).
- **Rozšířené úlohy** (*extended tasks*): tyto úlohy se od základních odlišují tím, že je při nich možné používat volání operačního systému *WaitEvent*, které uvádí úlohu do stavu čekání *waiting*. Tento stav umožňuje uvolnění procesoru a následně jeho přiřazení úloze s nižší prioritou, bez nutnosti ukončení běžící rozšířené úlohy.

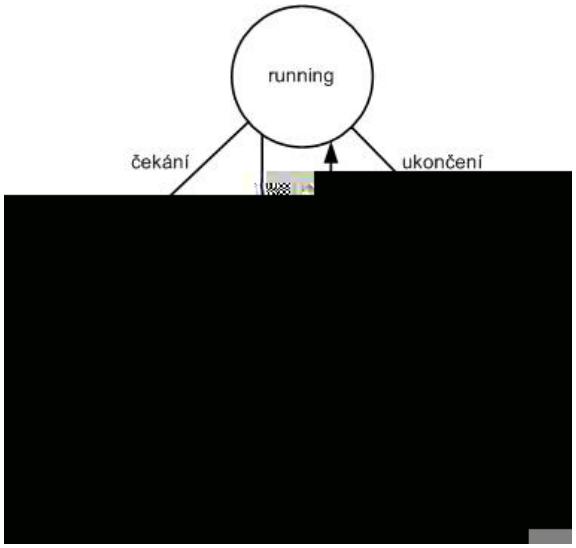
Vzhledem k operačnímu systému je řízení rozšířených úloh zpravidla složitější než řízení základních úloh a proto vyžaduje více systémových prostředků.

4.2.4.1 Stavový model úlohy

Úloha musí být schopna procházet mezi několika stavy, protože procesor může vždy vykonávat pouze jednu instrukci úlohy, přičemž ve stejnou dobu může být procesorem zpracováváno několik úloh. Operační systém OSEK je odpovědný za ukládání a obnovování kontextu úloh společně s přechody stavů úloh, kdykoli je to nutné.

4.2.4.1.1 Rozšířené úlohy: Rozšířené úlohy mají čtyři stavy (viz obrázek 4.6):

- **running**: v tomto stavu je úloha přiřazena CPU, takže je možné vykonávat její instrukce. V každém časovém okamžiku se může nacházet v tomto stavu pouze jedna úloha, zatímco ve všech ostatních stavech se může nacházet současně několik úloh.
- **ready**: v tomto stavu má úloha všechny funkční předpoklady pro přechod do stavu *running*, úloha pouze čeká na přidělení procesoru. Plánovač rozhoduje o tom, která z úloh ve stavu *ready* bude vykonávána jako další.
- **waiting**: úloha nemůže pokračovat ve vykonávání, protože musí čekat na nejméně jednu událost.
- **suspended**: v tomto stavu je úloha dočasně pasivní a může být aktivována.



Obrázek 4.6: Stavový model rozšířené úlohy.

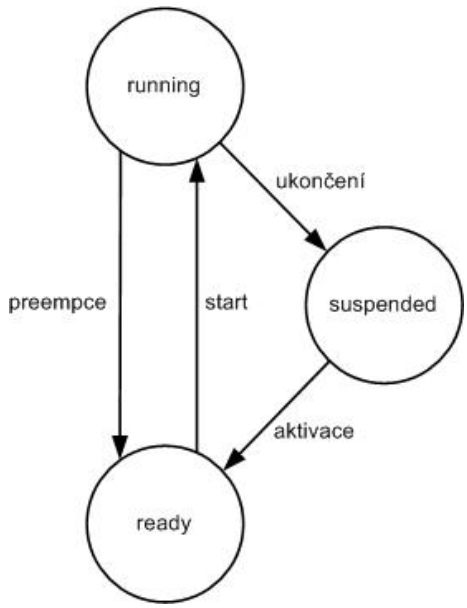
Přechod	Předchozí stav	Nový stav	Popis

Obrázek 4.7: Stavy a přechody rozšířených úloh.

Ukončení úlohy je možné v případě, že úloha ukončí sama sebe (*"self-termination"*). Toto omezení redukuje složitost operačního systému. Neexistuje žádný způsob jak přejít ze stavu *suspend* do stavu *waiting*. Tento přechod je nadbytečný a vedl by ke zvýšení složitosti plánovače.

4.2.4.1.2 Základní úlohy: stavový model základních úloh, jak ukazuje obrázek 4.8, je téměř identický jako stavový model rozšířených úloh. Jediný rozdíl je v tom, že základní úlohy nemají stav *waiting*.

- **running:** v tomto stavu je úloha přiřazena CPU, takže je možné vykonávat její instrukce. V každém časovém okamžiku se může nacházet v tomto stavu pouze jedna úloha, zatímco ve všech ostatních stavech se může nacházet současně několik úloh.
- **ready:** v tomto stavu má úloha všechny funkční předpoklady pro přechod do stavu *running*, úloha pouze čeká na přidělení procesoru. Plánovač rozhoduje o tom, která z úloh ve stavu *ready* bude vykonávána jako další.
- **suspended:** v tomto stavu je úloha dočasně pasivní a může být aktivována.



Obrázek 4.8: Stavový model základní úlohy.

Přechod	Předchozí stav	Nový stav	Popis
activate	*suspended*	*ready*	Nová úloha je nastavena do stavu *ready* službou systému. Nová úloha se stává aktivní.
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]

Obrázek 4.9: Stavy a přechody základních úloh.

4.2.4.1.3 Porovnání obou typů úloh: Základní úlohy nemají stav *waiting*, a proto obsahují synchronizační body pouze na začátku a na konci úlohy. Části aplikace s vnitřními

body synchronizace musí být implementovány použitím více než jedné základní úlohy. Výhoda základních úloh spočívá v tom, že požadují menší nároky na paměť RAM. Výhodou rozšířených úloh je, že umožňují sloučit souvislé činnosti do jedné úlohy. V případě, že chybí aktuální informace pro další zpracovávání, tak prodloužená úloha přejde do stavu *waiting* a vyčkává do doby, než tuto informaci získá a poté přechází opět do stavu *ready* nebo *running*. Rozšířené úlohy mají také více synchronizačních bodů než základní úlohy.

4.2.4.2 Aktivace úloh

K aktivaci úlohy dochází prostřednictvím služeb operačního systému *ActivateTask* nebo *ChainTask*. Jakmile dojde k aktivaci, začne se úloha vykonávat a to od svého začátku. Operační systém OSEK nepodporuje předávání parametru úloze v okamžiku startování úlohy. Tyto parametry mohou být předány pomocí zpráv nebo globálních proměnných.

4.2.4.2.1 Vícenásobná žádost o aktivaci úlohy: Základní úloha může být aktivovaná jednou nebo vícekrát v závislosti na třídě konformity. Vícenásobná žádost o aktivaci úlohy znamená, že OSEK operační systém přijímá a zaznamenává paralelní aktivace základní úlohy, která již byla aktivovaná.

Počet paralelních vícenásobných žádostí je definovaný u základní úlohy pomocí atributu při generování systému. Jestliže maximální počet vícenásobných žádostí není dosažen, je žádost zařazena do fronty. Žádosti o aktivaci základní úlohy jsou řazeny do fronty podle priority s seznamu aktivací.

4.2.4.3 Mechanismus přepínání úloh

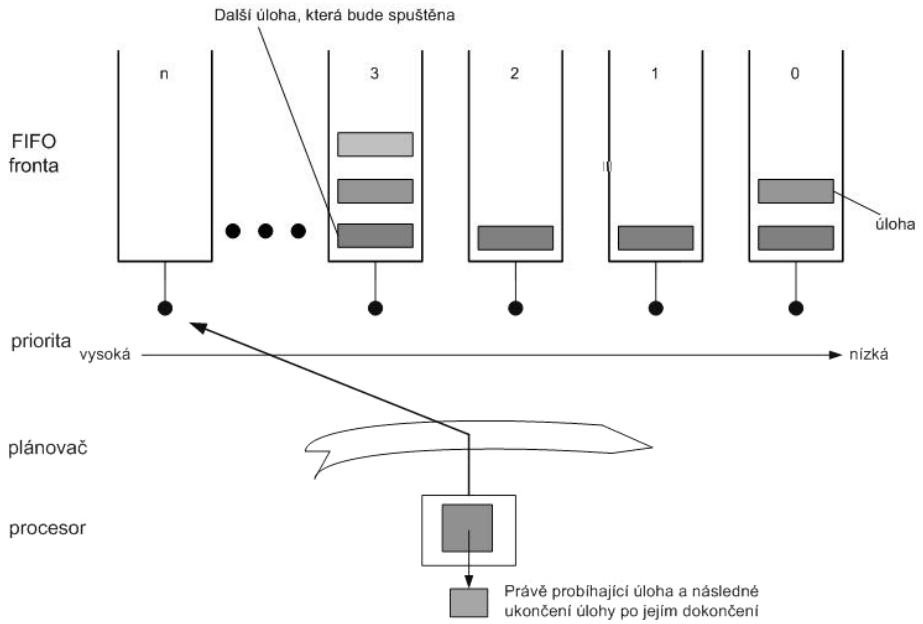
Operační systém dovoluje, na rozdíl od sekvenčního programování, zpracovávání více úloh současně, tzv. *multitasking*. Entita, která rozhoduje o tom, která úloha začne být vykonávána a která spouští všechny potřebné činnosti operačního systému, se nazývá plánovač. Plánovač je aktivován vždy když může dojít k přepnutí úlohy v závislosti na implementaci plánovací strategie. Plánovač můžeme chápat jako zdroj, který může být obsazen a následně uvolněn úlohami, tzn. že úloha může obsadit plánovač a zabránit tak přepnutí úlohy po dobu než jej opět uvolní.

4.2.4.4 Priorita úloh

Plánovač rozhoduje podle velikosti priority o tom, která *ready* úloha přejde do stavu *running*. Hodnota 0 definuje nejnižší hodnotu priority úlohy, každá jiná hodnota znamená vyšší prioritu úlohy.

Z důvodu zvýšení efektivity není definovaná dynamická správa priorit. Priorita úlohy je staticky definovaná uživatelem v průběhu generování systému a během vykonávání již nemůže být měněna.

Úlohy se stejnou prioritou jsou podporované pouze v BCC2 a ECC2 třídách konformity (viz obrázek 4.5, v takovém případě pořadí spouštění úlohy závisí na pořadí jejich aktivace. Rozšířené úlohy přitom ve stavu *waiting* neblokují spuštění úloh se stejnou prioritou.



Obrázek 4.10: Plánování.

Obrázek 4.10 ukazuje příklad implementace plánovače s použitím úrovně priorit. Ukazuje několik úloh s různými prioritami ve stavu *ready*, konkrétně jsou to tři úlohy s prioritou 3, jedna s prioritou 2, jedna s prioritou 1 a dvě s prioritou 0. Úloha, která čeká nejdéle závisí na tom, kdy podala žádost o aktivaci a na obrázku je vždy na konci každé fronty. Procesor právě zpracoval a ukončil úkol. Plánovač vybírá další úlohu, která bude spuštěna (tou se stává úloha s prioritou 3, a ta která je první ve frontě). Úloha s prioritou 2 bude spuštěna v okamžiku, až všechny úlohy s vyšší prioritou přejdou ze stavu *running* do stavu *ready*.

Pro stanovení, která další úloha bude prováděna jsou definované následující kroky:

- Plánovač vyhledá všechny úlohy, které jsou ve stavu *ready* nebo *running*.
- Z této množiny plánovač vybere úlohy s nejvyšší prioritou.
- Z této množiny nakonec plánovač vybere nejstarší čekající úlohu na procesor a ta se začne provádět.

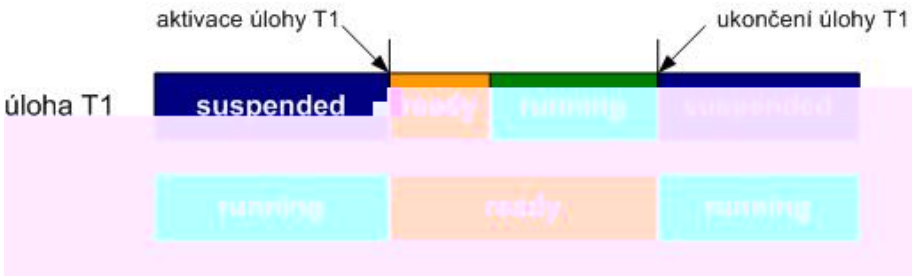
4.2.4.5 Plánovací strategie

4.2.4.5.1 Preemptivní plánování: preemptivní plánování znamená, že úloha která je právě ve stavu *running* může být přeplánovaná v libovolné instrukci úlohy v důsledku výskytu přednastavených spouštěcích podmínek operačním systémem. Preemptivní plánování přepne *ready* úlohu do stavu *running*, jakmile se úloha s nejvyšší prioritou dostane do stavu *ready*. Kontext úlohy je uložen, aby přerušená úloha mohla pokračovat v místě, kde byla přerušena.

U preemptivního plánování je doba čekání na spuštění úlohy s vyšší prioritou nezávislé na době vykonávání úloh s nižší prioritou. Tento způsob plánování vyžaduje větší nároky na paměťový prostor (RAM) v důsledku ukládání kontextu přerušené úlohy a větší složitosti při synchronizaci mezi úlohami. Každá úloha může být teoreticky přeplánována v

jakémkoliv místě, proto musí být zajištěna synchronizace přístupu k datům, která jsou užívány společně s ostatními úlohami.

Obrázek 4.11 ukazuje, že úloha T2 s nižší prioritou nepozdrží plánování úlohy T1 s vyšší prioritou.



Obrázek 4.11: Preemptivní plánování.

V případě preemptivního systému musí uživatel očekávat nucené přerušení běžící úlohy. Jestliže v části úlohy nesmí dojít k přerušení, tak může být zajištěno dočasné blokování plánovače pomocí systémové služby *GetResource*.

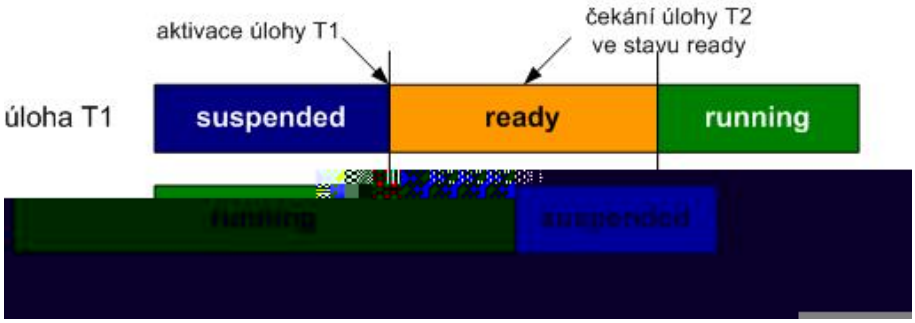
K přeplánování dojde v následujících případech:

- Úspěšné přerušení úlohy (pomocí služby *TerminateTask*).
- Úspěšné přerušení úlohy s explicitní aktivací následující úlohy (pomocí služby *ChainTask*).
- Aktivace úlohy na úrovni úloh (pomocí služby *ActivateTask*).
- Explicitní čekání volané v případě, že úloha přešla do stavu *waiting* (pouze pro rozšířené úlohy, pomocí služby *WaitEvent*).
- Nastavení události u úlohy ve stavu *waiting* na úrovní úloh (pomocí služby *SetEvent*).
- Vrácení zdrojů na úrovni úloh (pomocí služby *ReleaseResource*).
- Vrácení z úrovně přerušení do úrovně úloh.

Během obsluhy přerušení nedochází k žádnému přeplánování. Aplikace, které používají preemptivní plánování nepotřebují systémovou službu *Schedule*, pomocí které v místě jejího volání dojde k přeplánování. Ostatní plánovací strategie tuto službu mohou použít.

4.2.4.5.2 Nepreemptivní plánování: u tohoto typu plánování, může dojít k přepnutí úlohy pouze použitím explicitně definovaných systémových služeb (explicitní body přeplánování). Nepreemptivní plánování přináší zvláštní omezení na časové požadavky úloh. Specificky, nepreemptivní část úlohy, která je ve stavu *running*, úloha s nižší prioritou zpozdí start úlohy s vyšší prioritou až do dalšího bodu přeplánování.

Obrázek 4.12 ukazuje, že úloha T2 s nižší prioritou zpozdí úlohu T1 s vyšší prioritou do dalšího bodu přeplánování (v tomto případě ukončení úlohy T2).



Obrázek 4.12: Nepreemptivní plánování.

U nepreemptivního plánování může dojít k přeplánování v těchto případech:

- Úspěšné přerušení úlohy (pomocí služby *TerminateTask*).
- Úspěšné přerušení úlohy s explicitní aktivací následující úlohy (pomocí služby *ChainTask*).
- Explicitní volání plánovače (pomocí služby *Schedule*).
- Přechod úlohy do stavu *waiting* (pomocí služby *WaitEvent*).

Implementace nepreemptivních systémů může vést k tomu, že služby operačního systému, které způsobují přeplánování, mohou být volány pouze na vyšší programové úrovni a ne ve funkcích, které volají úlohy.

4.2.4.5.3 Smíšené plánování:

v této metodě plánování se vyskytují jak preemptivní, tak nepreemptivní úlohy. V tomto případě plánovací strategie závisí na preempčních vlastnostech běžící úlohy. Pokud běžící úloha je nepreemptivní, pak se provádí nepreemptivní plánování. Běží-li preemptivní úloha, pak se provádí preemptivní plánování. Definovat nepreemptivní úlohu v preemptivním operačním systému dává smysl:

- v případě, že doba vykonávání úlohy je srovnatelná s dobou pro přepnutí úlohy,
- jestliže je velikost paměti RAM, používaná pro ukládání kontextu úlohy, využívána ekonomicky,
- pokud úloha nesmí být přerušena.

4.2.4.6 Ukončování úloh

V operačním systému OSEK může být úloha ukončena pouze sama sebou. Operační systém OSEK poskytuje službu *ChainTask*, která zajistí aktivaci dané úlohy ihned po skončení právě běžící úlohy. V případě použití této služby uvnitř sebe sama dojde po ukončení úlohy k její opakované aktivaci a nově spouštěná úloha je přesunuta na konec fronty se stejnou prioritou.

Každá úloha musí být na svém konci ukončena prostřednictvím služby *ChainTask* nebo *TerminateTask*.

4.2.5 Aplikační módy

Aplikační módy jsou určeny k tomu, aby operační systém OSEK mohl přecházet pod různými módy činnosti. Minimální počet podporovaných aplikačních módů, který musí být u každé aplikace definován, je jedna. Ten je určen pro módy činnosti, které se vzájemně vylučují. Jakmile dojde k odstartování systému, tak není možné aplikační mód měnit.

4.2.5.1 Účel aplikačních módů

Mnoho ECU může spouštět nezávislé aplikace (jako je např. tovární test, programování flash paměti nebo normální operace). Aplikační mód je chápán jako způsob strukturování softwaru běžícího v ECU podle různých podmínek. Jedná se o čisté mechanismy pro vývoj úplně oddělených systémů. Typicky každý aplikační mód používá svoji vlastní množinu úloh, přerušovacích rutin (ISR), alarmů a časových podmínek, přesto neexistuje žádné omezení úloh nebo přerušovacích rutin (ISR) běžících v různých módech. Sdílení úloh, alarmů a přerušovacích rutin (ISR) mezi různými módy je doporučené v případě, že je znovu požadovaná stejná funkčnost. V průběhu generování systému a optimalizaci nám aplikační módy pomáhají redukovat počet objektů operačního systému, které je třeba brát v úvahu. V průběhu aplikace se aplikační mód nemění, k jeho změně dochází například při přepnutí z továrního testu do normálního z důvodu, že každý takovýto režim používá jiný aplikační mód, ve kterém byl navržen.

4.2.5.2 Nastavení aplikačních módů při startu systému

V automobilových aplikacích je jednou z nejdůležitějších věcí otázka bezpečnosti, proto nastavení aplikačních módů při startu systému je pro ECU z toho důvodu důležité, protože podmínka pro reset systému může přijít kdykoli během normální činnosti. Z tohoto důvodu by měla být změna aplikačního módu velmi rychlá. Volbu, který aplikační mód se použije, by měla rozhodnout nějaká jednoduchá podmínka, jako například změna stavu na pinu procesoru. Volba aplikačního módu musí být rozhodnuta ještě před zavedení jádra operačního systému. Tato část kódu, která se stará o výběr aplikačního módu je nepřenositelná. To by mělo zabránit používaní dlouhých startovacích procedur.

Volba aplikačního módu se při vývoji aplikace staticky definuje v OIL souboru (v tomto souboru je nadefinována celá aplikace), viz kapitola 4.5.1.

4.2.5.3 Podpora pro aplikační módy

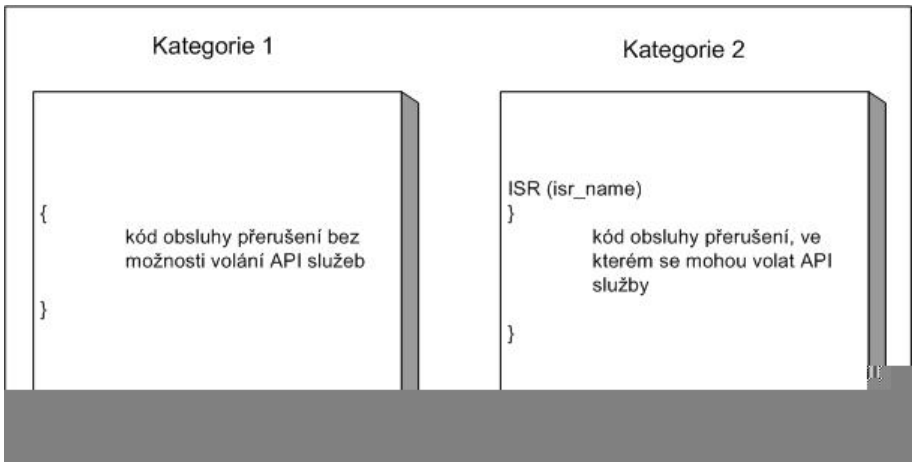
Neexistuje žádné omezení aplikačním módů pro podmnožiny tříd konformit. Aplikační módy jsou potřebné pro všechny třídy konformit. Přepínání aplikačních módů v průběhu vykonávání systému není dovoleno.

4.2.6 Zpracování přerušení

Funkce pro zpracování přerušení (služby operačního systému Interrupt Service Routines: ISR) jsou rozděleny do dvou ISR kategorií:

- **ISR kategorie 1** - ISR nepoužívají žádné služby operačního systému. Jakmile ISR skončí, dojde k pokračování přesně v tom místě, kde došlo k přerušení, tzn. že přerušení nemá žádný vliv na řízení úloh. ISR této kategorie mají minimální nároky.

- **ISR kategorie 2** - OSEK operační systém poskytuje ISR-rámce, které vytvářejí prostředí pro spouštění uživatelských rutin. V průběhu generování systému je těmto uživatelským rutinám přiřazeno přerušení. Tato kategorie ISR umožňuje používat některé služby operačního systému.



Obrázek 4.13: ISR kategorie operačního systému OSEK.

Uvnitř ISR se neprovádí žádné přeplánování. K přeplánování dojde v okamžiku, kdy je ukončena ISR kategorie 2, pokud byla přerušena preemptivní úloha a jestliže není aktivní žádné jiné přerušení. Implementace operačního systému OSEK zajistí, že úlohy jsou spouštěny podle plánovacích bodů (viz kapitola 4.2.4.5.1), aby mohlo být toto zajištěno, tak implementace operačního systému OSEK předepisuje omezení vztahující se na úrovně přerušení priorit pro ISR všech kategorií.

Maximální počet priorit přerušení závisí na použitém hardwaru a implementaci OSEKu. Plánování přerušení je závislé na hardwaru a ne na specifikaci OSEKu. Přerušení jsou plánována hardwarem, zatímco úlohy jsou plánovány plánovačem. Přerušení může přerušit úlohy (preemptivní nebo nepreemptivní). Pokud je úloha aktivovaná z přerušovacích rutin, tak k jejímu naplánování dojde až po skončení všech přerušovacích rutin. Možná omezení týkající se úrovně priorit přerušení jsou popsána v [6] (OS22.pdf).

4.2.7 Mechanismus událostí

Mechanismus událostí:

- Události můžou být použity jako prostředek pro synchronizaci úloh.
- Události jsou poskytovány pouze rozšířeným úlohám.
- Události inicializují stavové přechody úloh do a ze stavu *waiting*.

Události jsou objekty spravované operačním systémem. Nejedná se o nezávislé objekty, protože každá událost je vždy přiřazena nějaké své rozšířené úloze, která se stává jejím vlastníkem. Každá rozšířená úloha pak má definovaný určitý počet událostí. Událost je identifikovaná jménem jejího vlastníka (jménem úlohy) svým jménem (nebo maskou). V okamžiku, kdy je aktivována daná úloha, událost je operačním systémem smazána.

Události můžou být použity pro zasílání binární informace rozšířené úloze, které jsou přiděleny. Význam událostí je definovaný danou aplikací, např. mohou být použity jako signalizace vypršení časovače, dostupnosti zdrojů, příjmu zprávy atd.

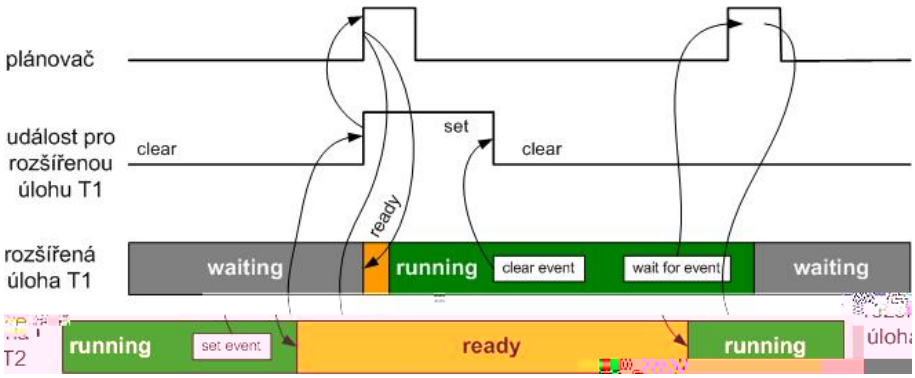
Operační systém OSEK definuje různé volby pro manipulaci s událostmi, v závislosti na tom, zda daná úloha je vlastníkem události nebo na jiné úloze, která nutně nemusí být rozšířenou úlohou. Všechny úlohy mohou nastavovat události pro rozšířené úlohy, které nejsou ve stavu *suspended*. Pouze vlastník události může smazat své události a čekat na nastavení svých událostí.

Události jsou kritéria pro přechod rozšířené úlohy ze stavu *waiting* do stavu *ready*. Operační systém poskytuje služby pro nastavení, mazání, dotazování a čekání na výskyt události. Nějaká úloha nebo ISR kategorie 2 mohou nastavit událost pro rozšířenou úlohu, která není ve stavu *suspended*. Tato událost pak informuje rozšířenou úlohu o změně stavu v místě, kde byla daná událost nastavena.

Příjemcem události může být pouze rozšířená úloha, proto není možné, aby ISR nebo základní úloha čekaly na nějakou událost. Rozšířené úlohy mohou smazat pouze události, které vlastní, zatímco základní úlohy nesmějí používat služby operačního systému pro mazání událostí.

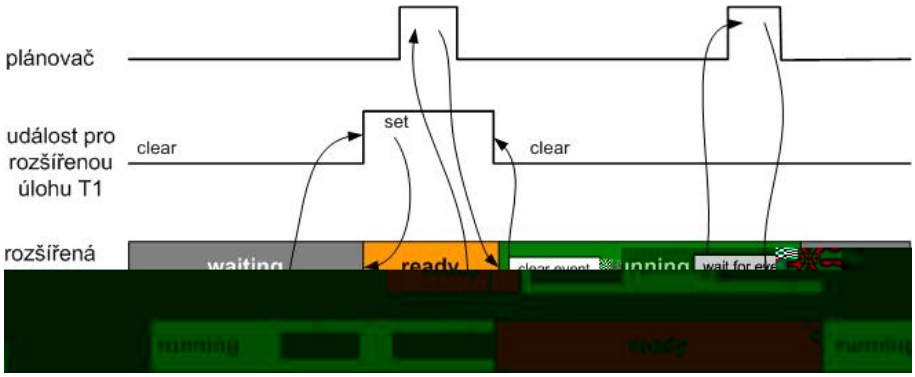
Rozšířená úloha ve stavu *waiting* je uvolněna do stavu *ready*, jestliže nastala alespoň jedna událost, na kterou úloha čekala. Pokud se pokusí rozšířená úloha ve stavu *running* čekat na událost. která již nastala, úloha setrvává ve stavu *running*.

Obrázek 4.14 ukazuje synchronizaci rozšířených úloh pomocí nastavení událostí v případě preemptivního plánování, kde rozšířená úloha T1 má nejvyšší prioritu. Úloha T1 čeká na událost. Úloha T2 nastavila tuto událost pro úlohu T1. Plánovač je aktivován. Dále, T1 přechází ze stavu *waiting* do stavu *ready*. Kvůli vyšší prioritě úlohy T1, dochází k přepnutí a úloha T2 je přerušena úlohou T1. T1 vymaže událost. Poté úloha T1 znovu čeká na událost a plánovač spouští pokračování úlohy T2.



Obrázek 4.14: Synchronizace preemptivních rozšířených úloh.

V případě nepreemptivního plánování, k přeplánování nedojde ihned po té, co došlo k nastavení události (viz obrázek 4.15, kde rozšířená úloha má vyšší prioritu).



Obrázek 4.15: Synchronizace nepreemptivních rozšířených úloh.

4.2.8 Správa zdrojů

Používá se pro řízení paralelního přístupu několika úloh s různými prioritami ke sdíleným zdrojům, jako jsou například plánovač, části programu nebo části hardwaru. Správa zdrojů je povinná pro všechny třídy konformity. Může být volitelně rozšířena pro řízení paralelního přístupu úloh a obsluh přerušovacích rutin ke sdíleným zdrojům.

Správa zdrojů zajišťuje, aby:

- dvěma úlohám nemohl být ve stejném čase přidělen stejný systémový zdroj.
- nedošlo k inverzi priorit.
- nedošlo k deadlocku při použití těchto systémových zdrojů.
- přístup k systémovým zdrojům nikdy neskončil ve stavu *waiting*.

Pokud je správa zdrojů rozšířena do přerušovací úrovně, musí být navíc zajištěno, aby dvěma úlohám nebo přerušovacím rutinám nemohl být přidělen ve stejném časovém okamžiku ten samý systémový zdroj.Případy použití správy zdrojů:

- V případě preemptivních úloh.
- V případě nepreemptivní úlohy, jestliže uživatel chce mít aplikační kód spustitelný pod jinou plánovací strategií.
- Sdílení zdrojů mezi dvěmi úlohami a přerušovací rutinou.
- Sdílení zdrojů mezi dvěmi přerušovacími rutinami.

Pokud uživatel požaduje ochranu proti možnému přerušení, tak může použít služby operačního systému pro povolení resp. zakázání přerušení, které nezpůsobují to, že dojde k přeplánování.

4.2.8.1 Chování během přístupu k obsazeným zdrojům

OSEK předepisuje tzv. *priority ceiling* protokol (protokol maximální dostupné priority), díky kterému nenastane situace, při které se úloha nebo přerušení pokouší dostat k obsazenému zdroji. Pokud je správa zdrojů použita pro koordinaci úloh a přerušení

operačního systému OSEK zajistí, že služby pro obsluhu přerušení jsou vykonávány pouze v případě, že všechny zdroje, které by mohly být použity pro obsluhu tohoto přerušení jsou volné. OSEK striktně zakazuje vnořený přístup ke stejným zdrojům.

4.2.8.2 Omezení použití systémových zdrojů

Služby operačního systému *TerminateTask*, *ChainTask*, *Schedule*, *WaitEvent* nesmí být volány dokud je zdroj obsazený nějakou úlohou. V případě vícenásobného přístupu ke zdroji v jedné úloze, musí uživatel požadovat a uvolňovat zdroje podle principu LIFO (Last In, First Out).

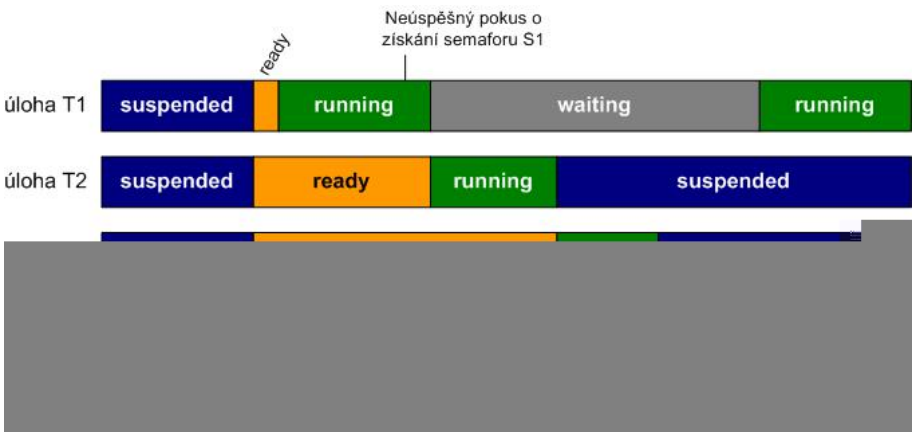
4.2.8.3 Plánovač jako systémový zdroj

Pokud musí úloha ochránit sama sebe proti nuceným přerušením jinou úlohou, může uzamknout plánovač. Plánovač je chápán jako systémový zdroj, který je přístupný pro všechny úlohy. Z tohoto důvodu je vygenerován zdroj s předdefinovaným jménem *RES_SCHEDULER*. Přerušení jsou přijímána a zpracovávána nezávisle na stavu systémového zdroje *RES_SCHEDULER*, který zabraňuje přeplánování úloh.

4.2.8.4 Problémy se synchronizačním mechanismem

4.2.8.4.1 Inverze priorit: Typickým problémem u běžných synchronizačních mechanismů, jako je použití semaforů, je problém týkající se inverze priorit. Tím je myšleno to, že úloha s nižší úrovní priority zdržuje vykonávání úlohy s vyšší úrovní priority, proto OSEK předepisuje již zmiňovaný *priority ceiling* protokol, který zabraňuje inverzi priorit.

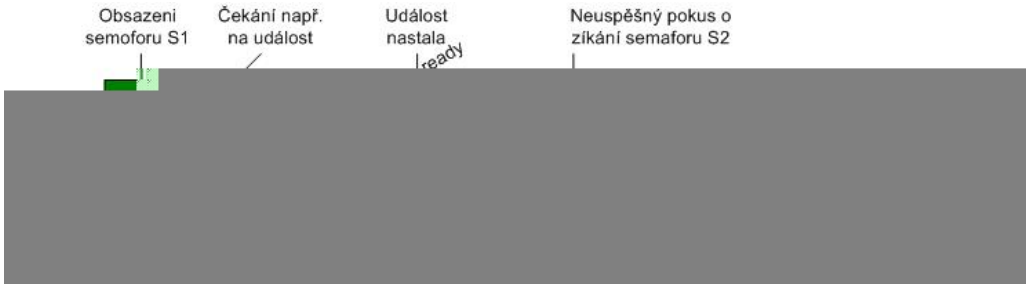
Obrázek 4.16 ukazuje řazení běžného přístupu dvou úloh k semaforu (v preemptivním systému, kde úloha T1 má nejvyšší prioritu). Úloha T4, která má nejmenší prioritu, vlastní semafor S1. Úloha T1 přeruší úlohu T4 a pokouší se získat stejný semafor, ale ten je obsazený, tak úloha T1 přejde do stavu *waiting*, ve kterém čeká na semafor S1. Nyní je úloha T4 přerušena úlohami s prioritou mezi T1 a T4. Úloha T1 může pokračovat až poté, co všechny úlohy s nižší prioritou budou dokončeny a semafor S1 bude znovu uvolněn. Přestože úlohy T2 a T3 nepoužívají semafor zdržují úlohu T1 v jejím vykonávání.



Obrázek 4.16: Inverze priorit na obsazeném semaforu.

4.2.8.4.2 Deadlocks: Jiným typickým problémem při běžných synchronizačních mechanismech, je problém deadlocků. V tomto případě deadlock znamená nemožnost vykonávání úlohy kvůli nekonečnému čekání na vzájemně blokované zdroje.

Následující obrázek 4.17 ukazuje vznik takového deadlocku. Úloha T1 obsazuje semafor S1 a následně nemůže pokračovat ve vykonávání, protože například čeká na nějakou událost. Tak úloha s nižší prioritou T2 je přepnuta do stavu *running* a pokouší se získat semafor S2, který získá. Jestliže se úloha T1 opět dostane do stavu *ready* a pokouší se získat semafor S2, tak se vrátí opět do stavu *waiting*, protože semafor S2 je již přidělen úloze T2. Procesor nyní získává úloha T2, která se snaží získat semafor S1, ale ten používá úloha T1 a celý systém se tak dostává do deadlocku.



Obrázek 4.17: Vznik deadlocku při použití semaforu.

4.2.8.5 Protokol maximální dostupné priority (*priority ceiling protocol*)

Aby nedocházelo k inverzi priorit a tím i k možnému vzniku deadlocku, operační systém OSEK vyžaduje následující chování:

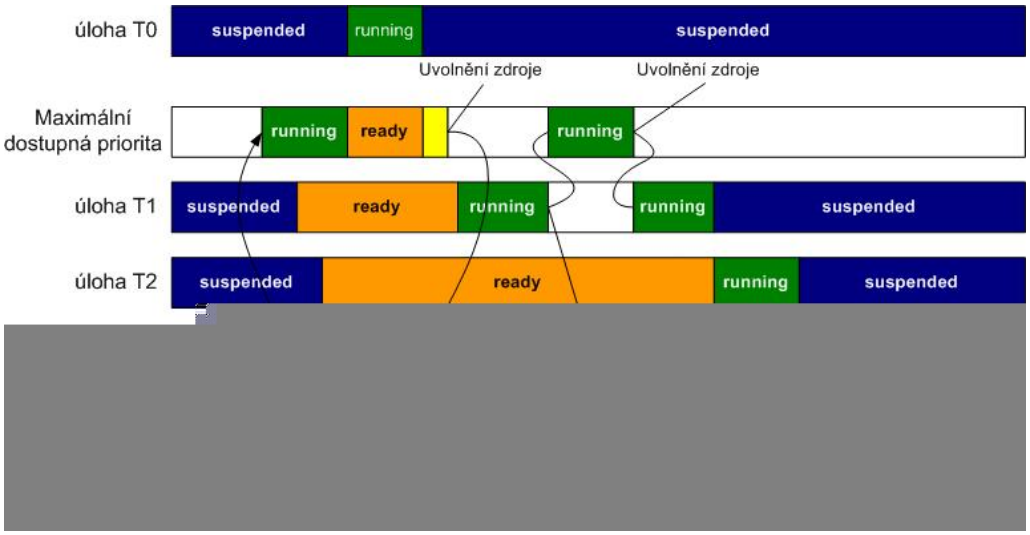
- V průběhu generování systému, je každému systémovému zdroji staticky přidělena vlastní maximální dostupná (*ceiling*) priorita. Tato hodnota bude nastavena alespoň na takovou úroveň, která odpovídá nejvyšší úrovni priorit všech úloh, které přistupují k danému systémovému zdroji. Zároveň tato hodnota musí být nižší něž je nejnižší úroveň priorit všech úloh, které k danému systémovému zdroji nepřistupují a které mají úroveň priority vyšší než nejvyšší priorita všech úloh, které přistupují k systémovému zdroji.

- Pokud úloha požaduje systémový zdroj a její současná úroveň priority je nižší než je maximální dostupná (*ceiling*) priorita systémového zdroje, tak priorita úlohy bude zvětšena na hodnotu maximální dostupné (*ceiling*) priority systémového zdroje.

- Pokud úloha uvolňuje systémový zdroj, bude její úroveň priority vrácena na hodnotu, kterou měla předtím něž požadovala systémový zdroj.

Maximální dostupná (*ceiling*) priorita má za následek možné zpoždění pro úlohy se stejnou nebo nižší prioritou, než je priorita systémového zdroje. Toto zpoždění je omezeno maximálním dobou, po kterou je daný zdroj obsazen úlohou s nižší prioritou. Úloha, která by mohla obsazovat stejný systémový zdroj jako právě běžící úloha, nepřejde do stavu (*running*), protože její priorita je nižší nebo rovna jako, je priorita právě běžící úlohy. Pokud systémový zdroj obsazený úlohou je uvolněn, tak další úloha, která by mohla daný

systémový zdroj obsadit přejde do stavu (*ceiling*). Pro preemptivní úlohy je toto bod pro přeplánování.



Obrázek 4.18: Přidělování zdrojů pomocí maximální dovolené priority mezi dvěmi preemptivními úlohami.

Obrázek 4.18 ukazuje mechanismus maximální dovolené priority. Úloha T0 má nejvyšší a úloha T4 nejnižší prioritu. Úloha T1 a T4 chtějí přistupovat ke stejnému systémovému zdroji. Systém jasně ukazuje, že nedochází k inverzi priorit. Úloha T1 s vysokou prioritou čeká kratší dobu, než je maximální povolená doba pro obsazení systémového zdroje úlohou T4.

4.2.8.6 Protokol maximální dostupné priority s rozšířením pro přerušovací úroveň

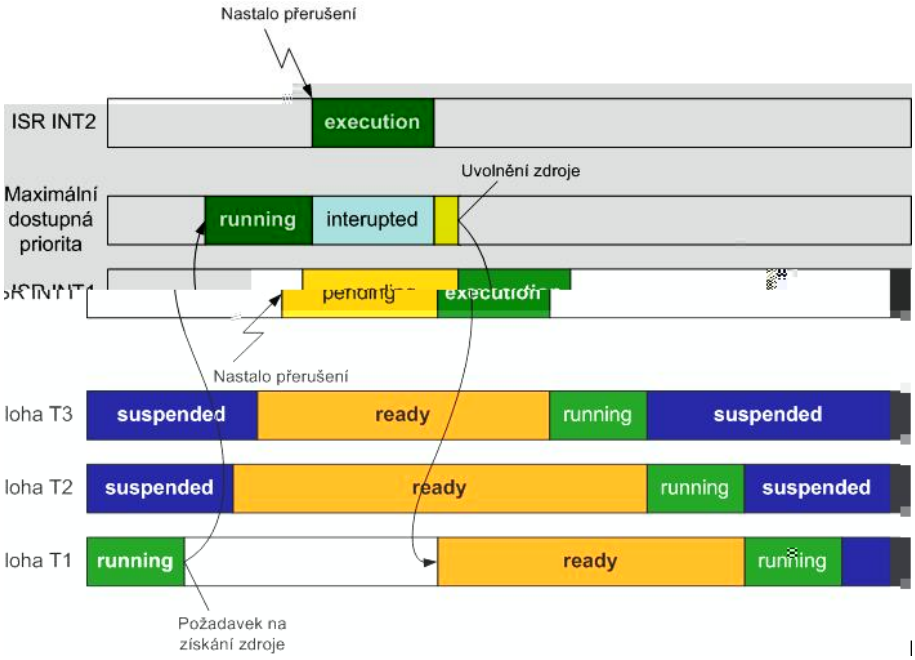
Rozšíření protokolu maximální dostupné priority i pro úroveň přerušení je nepovinné. Velikosti maximální dostupné priority systémového zdroje, který je používán v přerušení, je dána jako virtuální priorita vyšší, než jsou všechny priority úloh, kterým je přerušení přiřazeno. Zpracování softwarových priorit a hardwarových úrovní přerušení je závislé na konkrétní implementaci operačního systému.

- V průběhu generování systému je každému systémovému zdroji staticky přidělena maximální dostupná priorita. Hodnota maximální dostupné priority je alespoň taková, jako je nejvyšší hodnota priority ze všech úloh a přerušovacích rutin, které přistupují k systémovým zdrojům. Dále tato hodnota musí být nižší než nejnižší hodnota priorit všech úloh nebo přerušovacích rutin, které k danému systémovému zdroji nepřistupují a které mají ve stejném čase vyšší prioritu něž je nejvyšší priorita všech úloh nebo přerušovacích rutin, které přistupují k systémovým zdrojům.

- Pokud úloha nebo přerušovací rutina požaduje systémový zdroj a její současná úroveň priority je nižší než maximální dostupná priorita systémového zdroje, tak se její priorita zvýší na úroveň maximální dostupné priority systémového zdroje.

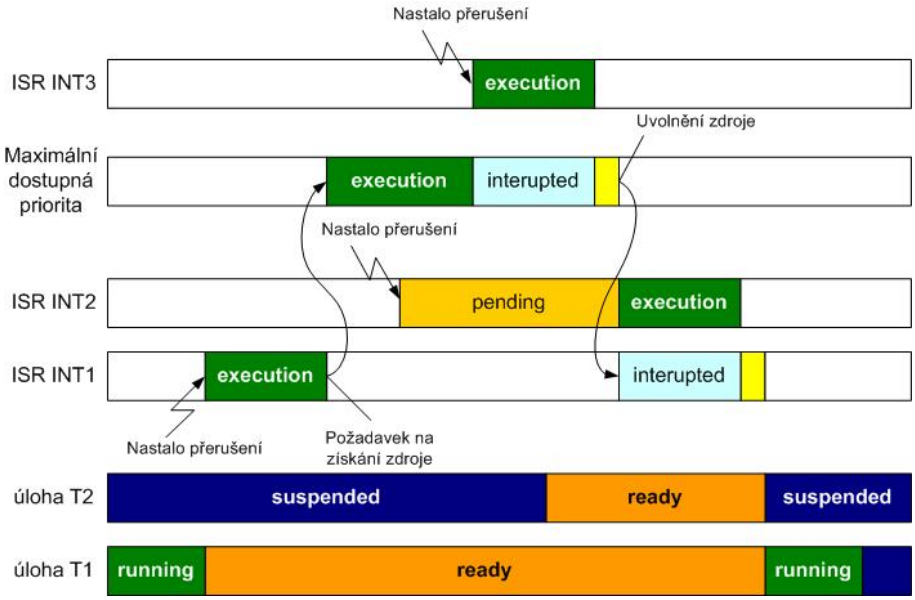
- Pokud úloha nebo přerušovací rutina uvolňuje systémový zdroj, je jejích priorita vrácena na hodnotu, kterou měly před tím, než žádaly o přidělení systémového zdroje.

Úlohy nebo přerušovací rutiny, které by mohly obsazovat stejný systémový zdroj, jako právě běžící úloha nebo přerušovací rutina, nepřejdou do stavu *running*, protože jejich priorita je nižší nebo rovna prioritě, kterou má právě běžící úloha nebo přerušovací rutina. Pokud je systémový zdroj obsazený úlohou uvolněn, tak další úloha nebo přerušovací rutina, která by mohla tento systémový zdroj obsadit, přejde do stavu *running*. Pro preemptivní úlohy je toto bod pro přeplánování.



Obrázek 4.19: Přidělování zdrojů pomocí maximální dovolené priority mezi dvěmi preemptivními úlohami a ISR.

Obrázek 4.19 ukazuje následující případ: Preemptivní úloha T1 je ve stavu *running* a požaduje systémový zdroj sdílený s ISR INT1. Úloha T1 aktivuje úlohy T2 a T3, které mají vyšší priority. Úloha T1 má nyní velikost priority rovnou hodnotě maximální dostupné priority a tudíž je vyšší než jsou priority úloh T2 a T3 a ty tudíž zůstávají ve stavu *ready*. Nastává přerušení INT1, ale jelikož úloha T1 stále běží díky momentálně nejvyšší prioritě (maximální dostupné), obsluha přerušení INT1 se ještě neprovede (vykonávání ISR je pozdrženo - pending). Nastává přerušení INT2. ISR INT2 přerušuje úlohu T1 a začne se vykonávat, po dokončení přerušení INT2 dochází k pokračování úlohy T1. Úloha T1 uvolňuje systémový zdroj a její priorita se vrací na svou původní hodnotu. Nyní je spuštěna obsluha přerušení INT1, protože má momentálně nejvyšší prioritu a úloha T1 je opět přerušena. Po skončení přerušení INT1 je spuštěna úloha T3. Po dokončení úlohy T3 se spouští úloha T2. Po jejím dokončení pokračuje vykonávání úlohy T1.



Obrázek 4.20: Přidělování zdrojů pomocí maximální dovolené priority mezi dvěmi ISR.

Obrázek 4.20 ukazuje následující případ: Preemptivní úloha T1 je ve stavu *running*. Nastává přerušení INT1. Úloha T1 je přerušena a je spuštěna obsluha přerušení INT1, které vyžaduje systémový zdroj sdílený s obsluhou přerušení INT2. Díky vyšší prioritě nastane přerušení INT2, ale protože přerušení INT1 má aktuálně vyšší prioritu díky protokolu maximálně dostupné priority, tak je stále ve stavu *running* a vykonávání přerušení INT2 je pozdrženo (pending). Nastává přerušení INT3, které díky vyšší prioritě než má přerušení INT1, přerušuje tuto obsluhu přerušení a je spuštěno. Přerušení INT3 aktivuje úlohu T2. Po skončení přerušení INT3, dochází k pokračování přerušení INT1. Poté co přerušení INT1 uvolní systémový zdroj, jeho priorita klesne na původní úroveň. Systémový zdroj nyní získá přerušení INT2, protože má momentálně v systému nejvyšší prioritu. Po skončení přerušení INT2, dochází opět k pokračování přerušení INT1. Po skončení přerušení INT1 úloha T2 přejde so stavu *running*, protože má vyšší prioritu než úloha T1, která je ve stavu *ready*. Po skončení úlohy T2, pokračuje úloha T1.

4.2.9 Alarmy (*Alarms*)

Operační systém OSEK poskytuje služby pro zpracování periodických událostí. Mezi takové události mohou například patřit časovač, který poskytuje přerušení v pravidelných intervalech nebo inkrementální snímač natočení, který generuje přerušení, když dojde k požadovanému natočení (např. vačkové nebo klikové hřídele) atd. OSEK poskytuje dvoustupňový koncept pro zpracování takovýchto událostí. Periodické události (zdroje) jsou registrované implementací specifikovanými čítači. Pomocí těchto čítačů OSEK nabízí pro aplikační software mechanismy alarmů.

4.2.9.1 Čítače

Čítače jsou reprezentovány hodnotou čítače, měřenou v "ticích" a specifikovanou konstantou čítače.

- Operační systém OSEK neposkytuje standardní API pro přímou práci s čítači.
- Operační systém OSEK se stará o nezbytné činnosti při správě alarmů, které vznikly rozšířením čítače.
- Operační systém OSEK nabízí alespoň jeden čítač, který je odvozen od (hardwarového nebo softwarového) časovače.

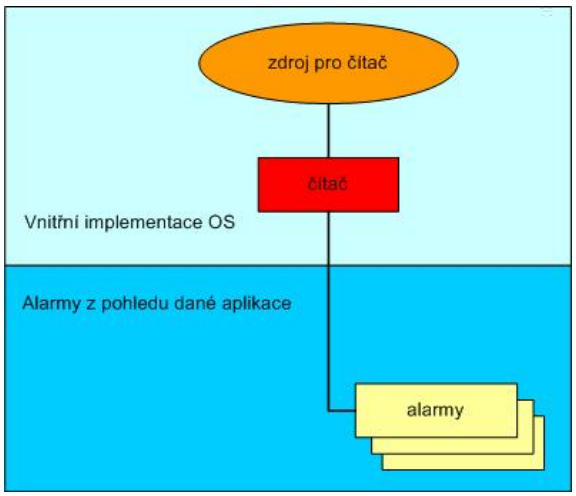
4.2.9.2 Správa alarmů

Operační systém OSEK poskytuje služby pro aktivaci úloh nebo nastavení událostí, které se provedou po vypršení alarmu. K vypršení alarmu dojde v okamžiku dosažení hodnoty, na kterou byl alarm nastaven. Tato hodnota může být definována relativně vzhledem k aktuální hodnotě čítače, pak se jedná o relativní alarm nebo jako absolutní hodnota čítače a pak se jedná o absolutní alarm. Alarmy mohou být definované jako jednoduché, ty po vypršení aktivují danou úlohu nebo nastaví událost a pak se vymaže nebo jako cyklické, které provedou danou akci vždy po vypršení (cyklicky). Operační systém navíc poskytuje služby pro smazání alarmu a získání aktuální hodnoty. Jeden čítač může být spojen s více alarmy.

Alarm je staticky přidělen v průběhu generování systému:

- jednomu čítači,
- jedné úloze.

Záleží na konfiguraci, co se má stát jakmile alarm vyprší.



Obrázek 4.21: Vrstvový model správy alarmů.

Čítače a alarmy jsou definované staticky, stejně tak jako přiřazení alarmů jednotlivým čítačům a akce které se mají provést po vypršení alarmu. Dynamickými parametry jsou, hodnota čítače při, které má dojít k vypršení a perioda pro cyklické alarmy.

4.2.10 Obsluha chyb, krokování a ladění

4.2.10.1 Hook rutiny

Operační systém OSEK poskytuje tzv. hook rutiny, které dovolují uživateli definovat akce, které se provádějí v době, kdy operační systém provádí své vnitřní aktivity, ke kterým jinak uživatel nemá přístup.

Tyto hook rutiny:

- jsou volané operačním systémem v určitých bodech činnosti v závislosti na implementaci operačního systému
- mají vyšší prioritu než všechny úlohy
- nejsou přerušitelné přerušovacími rutinami kategorie 2
- používají volání závislé na implementaci
- jsou součástí operačního systému
- jsou implementované uživatelem s definovanými funkcemi
- mají standardizované rozhraní při implementaci v operačním systému, ale jejich funkce standardizované nejsou a tudíž tyto rutiny nejsou přenositelné
- používají pouze omezenou množinu služeb operačního systému (viz [6] (OS22.pdf))
- jejich použití není povinné, dá se nastavit v OIL

V operačním systému OSEK jsou hook rutiny použity:

- při startování systému. To se používá hook rutina *StartupHook*, která se volá po zavedení operačního systému, ale před spuštěním plánovače
- při ukončování systému. To se používá hook rutina *ShutdownHook*, která se volá při ukončování systému. Tato rutina je požadována aplikací nebo operačním systémem v případě, že dojde ke kritické chybě.
- pro krokování nebo pro ladění programů
- pro obsluhu chyb

Každá implementace OSEKu popisuje zásady pro použití hook rutin. Pokud aplikace volá nedovolenou službu v hook rutinách, tak chování operačního systému není definované. Jestliže roste počet chyb, tak by daná implementace měla vrátit příslušný chybový kód. V hook rutinách není dovoleno používat většinu systémových služeb. Tímto omezením se snižuje složitost systému.

4.2.10.2 Obsluha chyb

Operační systém OSEK poskytuje službu pro obsluhu chyb, které mohou vzniknout při ladění aplikace nebo při běhu systému. Její základní rámec je předdefinovaný a musí být doplněn uživatelem. Toto dává uživateli systém pro efektivní centralizované nebo decentralizované zpracování chybových hlášení.

Rozlišují se dva druhy chyb:

- **Aplikační chyby:** operační systém nemůže vykonávat požadovanou službu správně, ale převezme správnost z jeho vnitřních dat. V tomto případě je volán centralizovaná obsluha chyb. Navíc operační systém vrací kód chyby pro decentralizovanou obsluhu chyb. Uživatel poté rozhodne co dělat na základě toho jaká chyba nastala.
- **Fatální chyby:** operační systém zjistil nesprávnost vnitřních dat. V takovém případě zavolá službu operačního systému, která jej ukončí.

Všem těmto chybám je přiřazen parametr, který specifikuje o jakou chybu se jedná.

4.2.11 Start systému

4. volání hook rutiny *StartupHook*, ve které uživatel definuje inicializační procedury. Během vykonávání hook rutiny jsou zakázána všechna přerušení.

5. Operační systém povoluje vykonávání uživatelských přerušení a startuje plánovací strategii. Spouštějí se úlohy a alarmy, které jsou nastaveny s parametrem *autostart* a jsou deklarovány pro aktuální aplikační mód.

4.2.11.1 Ukončení systému

Operační systém OSEK definuje speciální službu pro ukončení operačního systému, tzv. *ShutdownOS*. Tato služba může být požadována aplikací nebo operačním systémem při vzniku fatální chyby. Když je zavolána, tak operační systém nejprve volá hook rutinu *ShautdownHook* a pak ukončuje operační systém.

4.2.12 Ladění programu

Při přepínání úloh jsou volány dvě hook rutiny: *PreTaskHook a PostTaskHook* (viz obrázek 4.23). Tyto rutiny mohou být použity pro ladění programu nebo při měření času (včetně času, kdy došlo k přepnutí úloh). *PostTaskHook* rutina je volána po opuštění kontextu staré úlohy, *PreTaskHook* rutina je volána před vstupem kontextu nové úlohy. Při vykonávání těchto hook rutin, je úloha stále nebo právě ve stavu *running*, proto služba pro zjištění identifikátoru úlohy *GetTaskId*, nevrátí hodnotu *INVALID_TASK*.

Obrázek 4.23: Použití *PreTaskHook* a *PostTaskHook* rutin.

Jeli služba *ShutdownOS* volána v době, kdy úloha je ve stavu *running*, může nebo nemusí služba *ShutdownOS* volat *PostTaskHook* rutinu. Jestliže je *PostTaskHook* rutina zavolána, není definováno jestli je volána před nebo po *ShutdownHook* rutině.

4.3 Komunikace (*Communication - COM*)

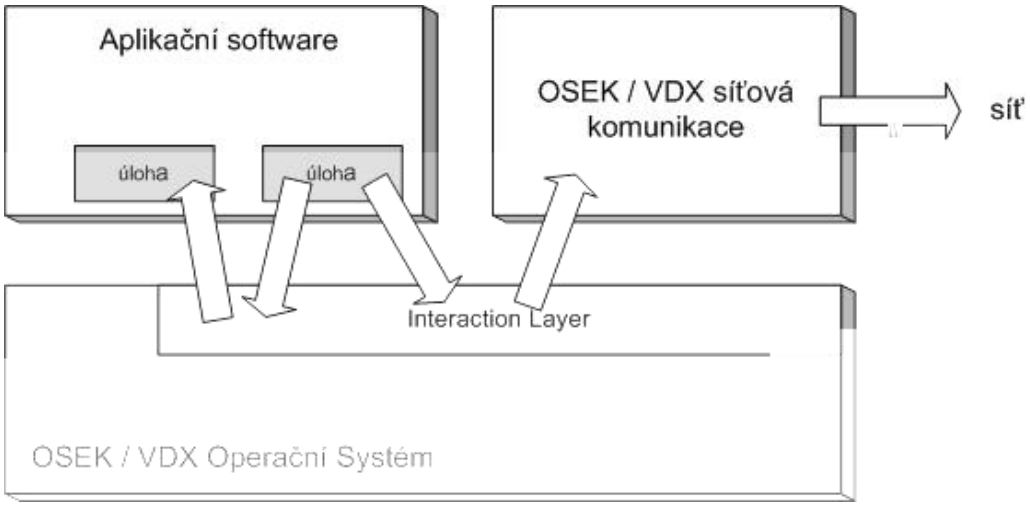
Materiály použité v této kapitole jsou čerpány z [6] (OSEK COM V3_0.pdf).

Specifikace OSEK/VDX Communication poskytuje rozhraní a protokoly pro vnitřní komunikaci ve vozidle. Tato komunikace zahrnuje jak komunikaci mezi propojenými ECU, tak i uvnitř těchto jednotek a komunikaci mezi ECU a periferií. Umístění Communication v OSEK/VDX architektuře ukazuje obrázek 4.1, zde je vidět jeho rozhraní s aplikací, s řízením sítě a s hardwarem přenosové sběrnice.

OSEK - COM je organizovaný do vrstev. Zahrnuje

Veškerá komunikace mezi úlohami se uskutečňuje výměnou zpráv, která je uložena v objektu zpráva *message*. OSEK/VDX komunikace představuje způsob volání zpráv, který rozlišuje dva druhy zpráv: *state messages* a *event messages*. *State message* reprezentuje hodnotu systémové proměnné, jako je např. teplota motoru, rychlost otáčení kol, atd. *State message* se neukládají do bufferu, ale jsou přepisovány aktuální hodnotou. Operace příjem čte hodnotu *state messages*. Na rozdíl *event message* obsahuje informaci o události, jako např. "teplota motoru překročila mezní hodnotu".*Event messages* se ukládají do bufferu s operací *send* a spotřebovány s operací *receive*.

Pro oba druhy komunikace, vnitřní a síťovou, se používá jednotné služby se shodným rozhraním. Na obrázku 4.24 je ukázána realizace. Služby a zprávy pro lokální komunikaci zprostředkovává *Interaction Layer*.



Obrázek 4.24: Model OSEK/VDX komunikace.

Oddělený modul pro síťovou komunikaci vykonává přenos zpráv po síti. Tento modul je vytvořený pomocí tasků, které užívají lokální komunikační služby. Každá zpráva je přenesena po síti jako místní zpráva, ta je vytvořena v každé stanici, která je zapojena do komunikace. Komunikační moduly síťové komunikace zajišťují to, aby zprávy přenášené uvnitř stanic, byly posílány po síti, v případě že existuje příjemce v další stanici. V přijímací stanici to zajišťuje to, že zprávy jsou aktualizovány společně.

Hlavní výhody OSEK/VDX komunikace jsou:

- Zajištění konzistence dat jako prevence proti konfliktům při paralelním přístupu tasků ke stejným datům.
- Přenositelnost je podporována jen pokud jsou OSEK/VDX komunikační služby použity pro komunikaci uvnitř tasků. Pak může být aplikace použita v jiných ECU.
- Vrstvová architektura OSEK/VDX komunikace (viz obrázek 4.1) dělá hlavní část komunikačního softwaru nezávislého na komunikačním protokolu (jako je např. CAN, VAN, ABUS, atd.) a komunikačním hardwaru.

4.3.1 Shrnutí OSEK COM

OSEK COM je zcela nezávislé komunikační rozhraní, které je možné použít v kombinaci i s jinými operačními systémy, které však vyhovují standardu OSEK. Programátor tak vlastně ani nepozná, jestli provádí komunikaci mezi úlohami v rámci toho samého procesoru nebo procesoru, který se nachází v úplně jiné řídící jednotce (ECU). Podmínkou pro komunikaci je pouze používání stejných identifikátorů zpráv, tzn. použít stejných označení zpráv při vysílání a při příjmu. O takovéto chování se právě stará komunikační rozhraní OSEK COM. Podmínkou, aby takovýto způsob komunikace mohl být realizován, je mít dostatečně velkou paměť RAM, tzn. že paměť která je integrovaná přímo v procesoru většinou nestačí, a proto je nutné použít jěště externí paměť RAM.

4.3.2 Realizace komunikace v operačním systému OSEKturbo

Jedná se o konkrétní realizaci operačního systému OSEK podle specifikace od společnosti Metrowrks. Tento konkrétní operační systém je určen pouze pro realizace aplikace OSEK na jednom procesoru. Komunikace v tomto operačním systému podporuje pouze CCCA komunikaci (*Communication Conformance Class A*), která zahrnuje pouze vnitřní komunikaci nefrontových zpráv, viz [7]. Komunikace je tedy možná pouze mezi úlohami a vyhovuje standardu OSEK COM v. 2.2.1.

Bližší informace o realizaci komunikace uvádí specifikace OSEK COM [6].

4.4 Správa sítě (*Network Management - NM*)

Materiály použité v této kapitole jsou čerpány z [6] (osek_nm251.pdf).

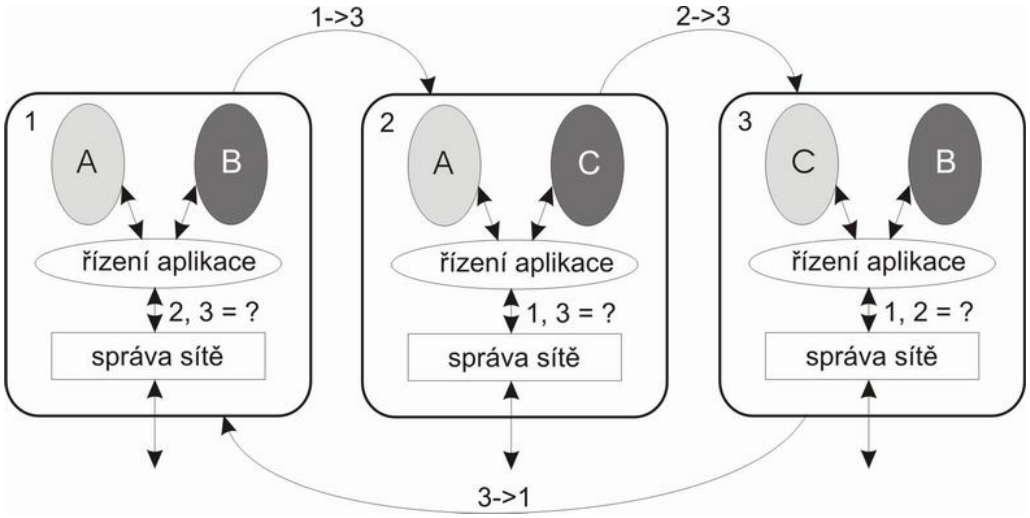
Architektury automobilových systémů se v současné době skládají z více či méně izolovaných elektronických jednotek, které jsou vzájemně propojeny do sítě a tvoří distribuovaný systém. Takto vzniklá síť tvoří základ distribuovaného řízení, které je nezávislé na lokálních ECU platformách. Důsledkem toho, chování lokální stanice zívisí na chování globální a naopak. Tyto vzájemné vlivy a závislosti často vyžadují rozsáhlou správu sítě (NM). Za účelem dát záruku a bezpečnost takovémuto distribuovanému systému, OSEK/VDX NM poskytuje podporu pro:

- Inicializaci síťové komunikace.
- Zabezpečení a monitorování síťové komunikace.
- Správu lokálních a globálních pracovních režimů stanic a sítě.
- Diagnostiky.

Základní koncept OSEK/VDX NM je monitorování jednotlivých stanic. NM poskytuje dva monitorovací mechanismy, tzv. *přímé* a *nepřímé* monitorování stanic. *Přímé monitorování* je založeno na trvalé komunikaci NM, tzn. že každé stanici se v pravidelných intervalech vysílá vlastní *alive-massage* (zpráva zda je stanice aktivní) a následně se od ostatních stanic přijímá, tím se zjistí, která ze stanic je aktivní či neaktivní. Synchronizace *alive-message* je zajištěna tím, že zprávy jsou posílány v logickém kruhu. *Nepřímé monitorování* stanic je založeno na poslouchání specifických zpráv při komunikaci aplikace. přijetí takovéto zprávy je chápáno jako *alive-message*.

Obrázek 4.25 ukazuje monitorování stanic. Aplikace A, B, C jsou všechny distribuovány mezi tři stanice. Každá aplikace potřebuje mít uvedeny všechny své podfunkce v síti jako

nezbytnou podmínku k zajištění své funkce. Tato podmínka musí být splněna během celého běhu aplikace. Dále jsou požadovány funkce monitorování, které mohou být realizovány s použitím monitorování stanic správou sítě, jestliže dostane mapovací funkce stanice.



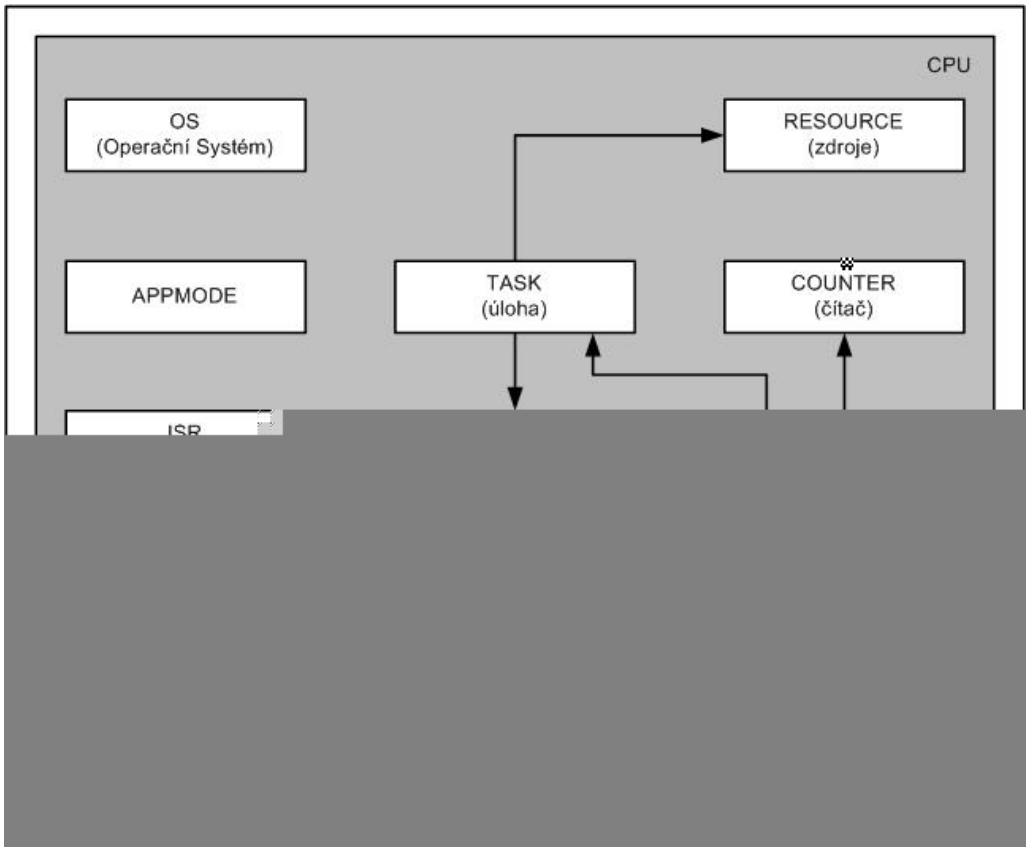
Obrázek 4.25: Monitorování stanic.

Bližší informace uvádí specifikace OSEK NM [6].

4.5 Implementační jazyk (*Osek Implementation Language - OIL*)

Materiály použité v této kapitole jsou čerpány z [6] (OIL23.pdf).

OIL je speciálně navržený jazyk pro vývoj embedded aplikací založených na OSEK konceptu. OIL se používá pro popis struktury aplikace pomocí souboru systémových objektů s definovanými odkazy. Úkolem OIL je tedy poskytnout konfiguraci OSEK aplikace pro dané CPU, tzn. že pro každé CPU se musí vytvořit vlastní OIL popis aplikace. Všechny OSEK systémové objekty (tasky, události, alarmy ...) jsou popsány pomocí OIL formátu. CPU slouží jako tzv. "nádoba" pro tyto systémové objekty. OIL definuje standardní typy pro systémové objekty (tasky, události, alarmy ...). Každé systémové objekty jsou definovány souborem atributů a odkazů. OIL explicitně definuje standardní atributy každého standardního systémového objektu. Možné odkazy definované v OIL pro standardní objekty ukazuje obrázek 4.26.



Obrázek 4.26: Standardní OIL objekty.

Každá OSEK implementace může definovat další specifické atributy a odkazy. K existujícím objektům lze přidávat pouze atributy. Vytváření nových objektů nebo změny v existujících objektech nejsou povoleny. Všechny nestandardní atributy (tzv. volitelné atributy) jsou považované za plnou specifickou implementaci a nemají žádný standardní výklad. Každá OSEK implementace může omezit existující soubor hodnot pro atributy objektů (např. omezí možnou hodnotu rozsahu priority).

4.5.1 OIL objekty a jejich atributy

4.5.1.1 CPU

Udává typ procesoru, na kterém běží aplikace pod OSEK OS řízením. Jedná se o speciální systémový objekt, který slouží jako "nádoba" pro lokální objekty, které slouží k popisu aplikací. Všechny aplikace jsou reprezentovány sadou systémových objektů, které jsou umístěny v CPU.

4.5.1.2 OS

Objekt OS (Operační Systém) je povinným objektem pro každou aplikaci. Slouží k definování vlastností OSEK operačního systému pro OSEK aplikaci. Proto pro každou aplikaci musí být definovaný jeden takovýto objekt. OIL nedefinuje v tomto systémovém objektu žádné standardní odkazy na další systémové objekty. Všechny ostatní systémové

objekty jsou řízené operačním systémem (OS). Operační systém má soubor globálních vlastností, pomocí kterých definuje chování aplikace. Některé z nich jsou specifické pro danou aplikaci, jiné jsou standardní. Mezi atributy OS ve standardní implementaci jsou považovány *Conformance Class* a *Extended Status* (viz OSEK OS specifikace [6]).

4.5.1.3 Atributy Operačního systému OS

- STATUS: atribut udává, zda musí být použit systém se standardním nebo rozšířeným stavem (*standard or extended status*). Tento atribut je ENUM s těmito možnými hodnotami:
  - STANDARD
  - EXTENDED
- HOOK routines: operační systém OS podporuje tyto atributy HOOK routines:
  - STARTUPHOOK
  - ERRORHOOK
  - SHUTDOWNHOOK
  - PRETASKHOOK
  - POSTTASKHOOK

  Jestliže se HOOK routines použijí, tak mají tyto atributy hodnotu TRUE, jinak je jejich hodnota FALSE.

4.5.1.4 APPMODE

Je to systémový objekt, který se používá k zavádění různých módů aplikace v jednom CPU. Objekt definuje OSEK OS vlastnosti pro OSEK OS aplikační mód. Pro tento objekt nejsou definované žádné standardní atributy. Pro každé CPU musí být tento objekt definován. Aplikační módy jsou navrženy k tomu, aby OSEK operační systém mohl přejít pod různé módy činnosti. Tyto módy povolují spuštění aplikace po ukončení systému v jednom z pevných módů s vlastním souborem úkolů.

4.5.1.5 ISR

Interrupt Service Routine (ISR) je zavolána, jakmile dojde k přerušení od nějakého zdroje jako je časovač nebo externí hadwarová událost. ISR mají nejvyšší prioritu, vyšší než všechny úlohy a plánovač. Adresy ISR ukazují do tabulky vektoru přerušení.

V operačním systému OSEK mohou ISR komunikovat s úlohami následujícími způsoby:

- ISR může aktivovat úlohu.
- ISR může poslat stav nebo zprávu o události úloze.
- ISR může spouštět časovač.
- ISR může získat stav úlohy.
- ISR může nastavit událost pro úlohu.
- ISR může ovládat alarmy.

4.5.1.6 Atributy systémového objektu ISR

- CATEGORY: tento atribut definuje kategorii ISR, je UNIT32 a přípustné hodnoty jsou 1, 2 a 3.
- RESOURCE: zde se definuje seznam zdrojů zpřístupněných ISR.
- ACCESSOR: používá se k definování odkazů k odeslání nebo přijetí zprávy (SENT or RECEIVE message).
  - SENT - zpráva bude ISR odeslaná.
  - RECEIVE - zpráva bude ISR přijatá.

  Dále je nutné zadefinovat parametr ACCESSNAME. Tento parametr může být použit aplikací k přístupu k datům ve zprávě a je typu *string*.

4.5.1.7 RESOURCE

Jedná se o systémový objekt, který se používá ke koordinaci několika úloh, které současně přistupují ke společným zdrojům, jako je např. plánovač, část programu, paměť nebo hadwarové zařízení. Každý zdroj může být v danou chvíli přiřazen pouze jedné úloze. Zdroj, který je právě přidělen úloze, musí být před přidělením další úloze uvolněn. V operačním systému OSEK jsou systémové prostředky řazeny podle priority. Každý zdroj má svou vlastní prioritu (*resource Ceiling Priority)*, která je staticky přidělena uživatelem. *Priority ceiling* protokol zvedne žádost úlohy o zdroj k úrovni priority. Tato priorita může být vypočítaná během chodu. *Ceiling* priorita je:

1. Stejná nebo vyšší než je nejvyšší priorita úlohy s přístupem ke zdroji (úloha T1).
2. Nižší než všechny ostatní úlohy o vyšší prioritě než úloha T1 (např. úloha T0).

V případě, kdy úloha obsazuje zdroj, systém dočasně změní prioritu úlohy na hodnotu Ceiling priority zdroje. Každá jiná úloha, která by mohla obsadit stejný zdroj, nemůže přejít do *running* stavu kvůli nižší nebo stejné hodnotě priority. Když je zdroj uvolněn, tak se úroveň priority úlohy vrací na původní hodnotu. Ostatní úlohy, které by mohly obsadit tento zdroj, nyní mohou přejít do *running* stavu. Za povšimnutí stojí, že je možné mít zdroje se stejnou prioritou.

Pro tento systémový objekt nejsou definované žádné atributy.

4.5.1.8 TASK

Obecně řídící software může být rozdělen na části vykonávané podle jejich požadavků reálného času. Tyto části mohou být implementovány přes tzv. úlohy (*TASKs*). Každá takováto úloha provádí určitou funkci celé aplikace. OSEK OS poskytuje paralelní a asynchronní spouštění úlohy, o které se stará plánovač (*SCHEDULER*). OSEK poskytuje dva typy úloh:

- Basic task (základní úloha) - takováto úloha uvolní procesor pouze v případě, že úloha je zrušena, že došlo k přepnutí úlohy s vyšší prioritou nebo že dojde k obsluze přerušení.

- **Extended task (rozšířená úloha)** - od Basic task se odlišují tím, že smějí používat volání operačního systému *WaitEvent*, které zavádí stav čekání. V tomto stavu dojde k uvolnění procesoru, který může být přiřazen úloze s nižší prioritou, aniž by došlo k ukončení běžící rozšířené úlohy.

Úloha musí být schopna se přepínat mezi několika stavy, protože procesor může v daném okamžiku vykonávat pouze jednu instrukci úlohy, zatím co o procesor v tom samém čase může soutěžit několik úloh. OSEK OS je zodpovědný za ukládání a obnovování úloh v souvislosti s přechody úloh mezi jednotlivými stavy. Úloha se může nacházet v jednom z následujících stavů:

- **Running** - v tomto stavu je procesoru přidělena úloha a procesor vykonává její instrukce. V daném okamžiku se v tomto stavu může nacházet pouze jedna úloha, kdežto v ostatních stavech se může nacházet několik úloh najednou.
- **Ready** - úloha má v tomto stavu všechny předpoklady, aby mohla přejít do *running* stavu, pouze čeká na přidělení procesoru. Plánovač rozhodne, která *ready* úloha bude vykonávána jako další.
- **Suspended** - v tomto stavu je úloha nečinná a může být aktivovaná.
- **Waiting** - úloha nemůže být dále vykonávána, protože čeká alespoň na jednu událost.

Extended úlohy se mohou nacházet ve všech čtyřech stavech, zatímco basic úlohy se mohou nacházet pouze ve třech (running, ready, suspend).

Úloha je aktivovaná, když přejde ze stavu *suspended* do stavu *ready*. Jakmile přejde do stavu *ready*, tak se ji musí začít přidělovat systémové zdroje (tzn. dynamické přidělování zásobníku a řídícího bloku úlohy (task node)). Basic úloha může být aktivována jednou nebo vícekrát. Počet vícenásobných souběžných žádostí o aktivaci je stanoven příslušným atributem během generování systému. V případě, že počet žádostí přesáhne maximální hodnotu, dochází k jejich řazení do fronty podle priority.

Úloha může být aktivovaná jinými úlohami nebo při startování systému. Po zavedení systému je spouštění úlohy plánováno podle úrovně její priority. V OSEK OS je priorita staticky přidělena každé úloze a za běhu nemůže být uživatelem měněna. Úloha může být buď:

- **Preemptivní** - takováto úloha může přejít ze stavu *running* do stavu *ready* kdykoli úloha s vyšší prioritou přejde do stavu *running*.
- **NE-preemptivní** - takováto úloha může přejít ze stavu *running* do některého jiného stavu pouze přes volání plánovače.

4.5.1.9 Atributy systémového objektu TASK

- **PRIORITY:** v tomto atributu se definuje hodnota priority. OSEK definuje nejnižší prioritu jako 0, každá vyšší hodnota tohoto atributu odpovídá vyšší prioritě.
- **SCHEDULE:** tento atribut udává, zda úloha bude preemptivní nebo nepreemptivní, může tedy nabývat hodnoty **FULL** pro preemptivní úlohu, nebo **NON** pro nepreemptivní úlohu.

- ACTIVATION: tento atribut definuje maximální počet žádostí pro úlohu, např. je-li tato hodnota 1, tak to znamená, že je povolena pouze jedna aktivace pro danou úlohu.

  Poznámka: v operačním systému OSEKturbo od společnosti Metrowerks může hodnota tohoto atributu nabývat pouze hodnoty 1.

- AUTOSTART: hodnota tohoto atributu může nabývat hodnot *TRUE* nebo *FALSE* a udává, zda má být daná úloha aktivována během startovací procedury systému, nebo ne.

- RESOURCE: v tomto atributu se definuje seznam zdrojů, ke kterým daná úloha přistupuje. Jedná se o vícenásobný odkaz typu *RESOURCE_TYPE*.

- EVENT: tento odkaz je používán pro definování seznamu událostí možných reakcí rozšířených úloh. Jedná se o vícenásobný odkaz typu *EVENT_TYPE*.

- ACCESSOR: tento atribut definuje vícenásobné odkazy pro odesílání a příjem zprávy. Dále se zde ještě navíc definuje parametr WITHOUTCOPY a ACCSESSNAME.

  - ACCESSOR = SENT - tento parametr definuje, že zpráva bude odeslána úlohou.
  - ACCESSOR = RECEIVE - tento parametr definuje, že zpráva bude přijata úlohou.

  Parametr WITHOUTCOPY specifikuje zda bude lokální kopie zprávy použita a je typu boolean. ACCESSNAME parametr definuje odkaz, který může být použit aplikací k přístupu k datům zprávy a je typu string.

4.5.1.10 ALARM

OSEK operační systém poskytuje služby pro aktivaci úloh nebo množiny událostí v okamžiku, kdy dojde k vypršení alarmu. Alarm vyprší v okamžiku dosažení předdefinované hodnoty čítače. Tato hodnota může být definovaná relativně vzhledem k aktuální hodnotě čítače, v takovém případě mluvíme o tzv. *relativních alarmech* (relative alarm), nebo jako absolutní hodnota čítače a pak mluvíme o tzv. *absolutních alarmech* (absolute alarm). Alarmy mohou být definované buď jako jednoduché (single alarms), nebo jako cyklické (cyclic alarms). Jednoduché alarmy aktivují úlohy nebo události pouze jednou po svém vypršení, kdežto cyklické alarmy aktivují úlohy nebo události cyklicky vždy po svém vypršení. Alarmy mohou také například přijímat potvrzení o počtu přerušení vyvolaných časovačem nebo přijímání zpráv. Dále OS poskytuje služby pro vymazání alarmů a služby pro získání aktuální hodnoty alarmů. Na jeden čítač může být napojeno i více alarmů. Alarm je staticky přiřazen vždy při generování systému:

- jednomu čítači.
- jedné úloze.

V závislosti na konfiguraci bude po vypršení alarmu aktivována úloha nebo nastavena událost pro tuto úlohu.

Čítače a alarmy jsou definované staticky. Přiřazení alarmů čítačům a akce, které se mají provést po vypršení alarmů, jsou definované také staticky. Dynamickými parametry jsou hodnota čítače, při které má dojít k vypršení čítače a perioda pro cyklické alarmy.

4.5.1.11 Atributy systémového objektu ALARM

- COUNTER: tento odkaz definuje čítač, který bude přiřazen alarmu. Alarmu může být přiřazen pouze jeden čítač, kdežto jednomu čítači může být přiřazeno více alarmů.
- ACTION: zde se definuje, která akce bude po vypršení alarmu provedena. Může nabývat těchto hodnot:
  - ACTIVATETASK: s tímto parametrem bude po vypršení alarmu aktivována úloha.
  - SETEVENT: s touto hodnotou dojde k nastavení události pro danou úlohu.

4.5.1.12 COUNTER

Čítač je reprezentován hodnotou čítače měřenou v "ticích" a konstantami, které definují parametry čítače. OSEK OS neposkytuje standardizované API pro přímou manipulaci s čítači. OSEK OS nabízí alespoň jeden čítač odvozený od (hardwarového nebo softwarového) časovače.

4.5.1.13 Atributy systémového objektu COUNTER

- MAXALLOWEDVALUE: tento atribut definuje maximální možnou hodnotu čítače.
- TICKSPERBASE: tento atribut specifikuje počet tiků potřebných pro dosažení specifické jednotky. Výklad je daný implementací.
- MINCYCLE: definuje minimální dovolený počet tiků čítače pro cyklické alarmy napojené na tento čítač.

4.5.1.14 EVENT

Jedná se systémový objekt, který je řízen OSEK OS a umožňuje ukládat binární data. Výklad objektu EVENT je dán až na uživateli, může být např. použit pro signalizování vypršení časovače, k zjištění dostupnosti zdroje, k potvrzení zprávy atd. Správa úloh slouží jako další OSEK OS mechanismus pro synchronizaci úloh.

OSEK OS události jsou přiděleny pouze rozšířeným úlohám, tzn. že mají povolení k přechodu do stavu *waiting* až do nějaké nastalé události (to je definované úlohou). Tak může rozšířená úloha pozdržet sama sebe. Každá úloha (základní nebo rozšířená) může oznámit rozšířené úloze ve stavu *waiting*, že nastala specifikovaná událost. Po tom se úloha stává připravená (ready) a stává se částí plánovacího procesu.

4.5.1.15 Atributy systémového objektu EVENT

- MASK: maska události je celé číslo typu UNIT64. Další způsob, jakým se dá přiřadit maska události, je AUTO, v tomto případě je jeden bit automaticky přiřazen masce událostí. Tento bit je unikátní s ohledem na úlohy, které se odkazují na události.

4.5.1.16 MESSAGE

V OSEK OS se komunikace mezi jednotlivými úlohami uskutečňuje pomocí zpráv. Způsob komunikace a správa systémových služeb pro zprávy je založena na OSEK COM (communication) v.2.2.1 z roku 2000. Zprávy jsou uložené v objektu MESSAGE a jsou

řízené operačním systémem. Zpráva může mít pouze jednoho odesílatele, ale může mít více příjemců (tzn., že tutéž zprávu může přijímat více objektů). Formát zprávy je pevně definován při generování systému. Zprávy mohou být dvou typů:

- Unqueued Messages (nefrontové zprávy): takováto zpráva reprezentuje aktuální hodnotu systémové proměnné, jako je např. teplota motoru, otáčky kola atd. Operace SEND přepíše aktuální hodnotu zprávy, tzn. že nefrontové zprávy se neukládají do bufferu. Operace RECEIVE čte aktuální hodnotu nefrontové zprávy, ta může být čtena z více míst aplikace.
- Queued Messages (frontové zprávy): frontová správa obsahuje informace o události, jako je např. informace o tom, že teplota motoru překročila mezní hodnotu. Frontové zprávy se ukládají do bufferu typu FIFO (first in first out), tzn. že jsou čteny v pořadí v jakým byly odeslány. Každá takováto zpráva může být čtena pouze jednou, protože operace čtení vyjme tuto zprávu z fronty.

4.5.1.17 Atributy systémového objektu MESSAGE

- TYPE: tento atribut definuje, zda je se jedná o frontovou či nefrontovou zprávu.

  Poznámka: operační systém OSEKturbo od společnosti Metrowerks podporuje pouze nefrontové (UNQUEUED) zprávy.

- CDATATYPE: tento atribut popisuje jakého typu jsou data ve zprávě, používá stejných typů jako jazyk C (např. *int*nebo jméno struktury).
- ACTION: tento atribut definuje co se stane, když se daná zpráva přijme. Může nabývat těchto hodnot:
  - NONE: odesláním zprávy s tímto parametrem se neprovede žádná akce.
  - ACTIVATETASK: odesláním zprávy s tímto parametrem dojde k aktivaci dané úlohy.
  - SETEVENT: tento parametr definuje úlohu, pro kterou byla nastavena událost. Odesláním zprávy s tímto parametrem se nastaví definovaná událost.
  - CALLBACK: tento parametr definuje funkci, která nastane odeslání takovéto zprávy.
  - FLAG: v tomto parametru se definuje jméno FLAGu, který se nastaví, když se zpráva odešle.

  Pro nefrontové zprávy je povolen seznam akci podle specifikace OSEK COM 2.2. Pro frontové zprávy je povolena pouze jedna akce.

4.5.1.18 COM

Tento systémový objekt reprezentuje lokální parametry vlastností OSEK komunikačního podsystému na CPU. OSEK komunikační koncept je používán jak OSEK OS, tak i OSEK COM. Lokální parametry komunikace a správy sítě jsou definované pro každé CPU zvlášť použitím speciálních systémových OIL objektů COM a NM. Společně se systémovým objektem OS plně definují OSEK systém na daném CPU. Pro jedno CPU smí být definovaný pouze jeden systémový objekt COM.

4.5.1.19 Atributy systémového objektu COM

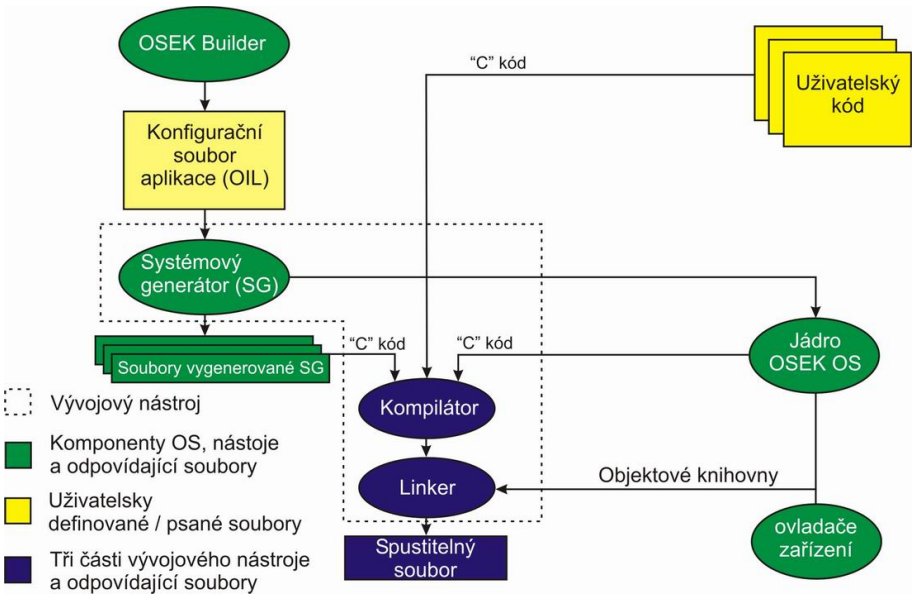
- USEMESSAGERESOURCE: tento atribut specifikuje, zda je použit mechanismus zdroje zpráv.
- USEMESSAGESTATUS: tento atribut specifikuje, jestli je status zprávy přístupný.

4.5.1.20 NM

Tento systémový objekt reprezentuje parametry správy sítě na daném CPU. Pro tento objekt nejsou definované žádné atributy.

4.5.2 Vývoj OSEK/VDX aplikace

Poslední verze OSEK systému (2.2 a 2.3) umožňují adresovat pouze jedno CPU v elektronické řídící jednotce (ECU) a ne celou síť ECU. Na obrázku 4.27 je uveden příklad vývojových procesů pro vytvoření OSEK aplikace. K vytvoření OIL popisu aplikace můžeme použít software, který vygeneruje OIL soubor s popisem aplikace. Takovým to softwarem je například OSEK Builder od společnosti Metrowerks. OIL soubor s popisem aplikace je možné také vytvořit "ručně", editováním tohoto OIL souboru (struktura OIL souboru viz implementace OIL 2.x [6]).



Obrázek 4.27: Vývoj OSEK/VDX aplikace.

Kapitola 5

Vývojové prostředí CodeWarrior a OSEK Builder

5.1 CodeWarrior

CodeWarrior je vývojové prostředí od společnosti Metrowerks, která je součástí společnosti Motorola a slouží k programování procesorů Motorola řady HC08, HC12 a PowerPC.

5.1.1 Hlavní funkce vývojového prostředí CodeWarrior

- Vysoce optimalizovaný C/C++ kompilátor s širokou možností jeho nastavení.
- Výkonné makro pro zápis kódu v Assembleru.
- SmartLinker, který slinkuje pouze pouze požadované objekty.
- Nástroj, tzv. *Burner*, na tvorbu výstupního souboru ve formátu: Motorola S-záznam, Intel Hex formát nebo jako Binární soubor. S tímto nástrojem je možné se pomocí sériového rozhraní PC přímo spojit s určitými typy podporovaného cílového HW a provést zápis programu programu do procesoru.
- Dekodér k dekódování objektů a tzv. absolutních souborů (to jsou soubory, ze kterých se následně generuje požadovaný typ výstupního souboru).
- Libmaker, nástroj pro vytváření knihoven.
- Víceúčelový debuger, který umožňuje provádět simulaci a pro podporované typy cílového hardwaru provádět real-time ladění programu přímo na reálném procesoru.
- Ladění programu, který je psán ve více jazycích, jako je assembler, C a C++.
- Plně funkční simulátor daného procesoru, ve kterém je možné odladit skutečnou aplikaci pro daný typ procesoru.

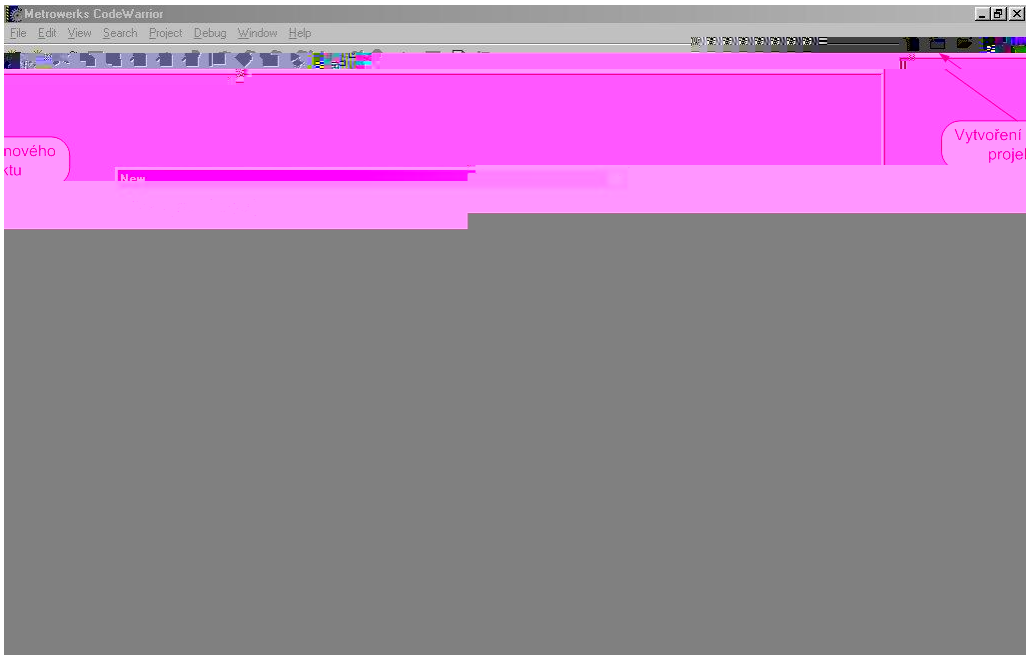
5.1.2 Založení nového projektu

Založení nového projektu se provádí standardním způsobem známým i z jiných vývojových nástrojů. Postup je následující:

1. Spuštění programu CodeWarrior ADS V2 z nabídky *Start* systému Windows.

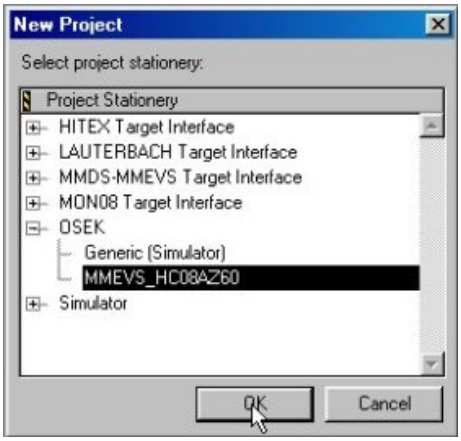
2. Kliknutím v menu na *File*, dále *New* a následuje volba: prázdný projekt, import projektu, nebo projekt připravený pro procesory řady HC08 nebo HC12. Druhou možností je kliknutí na tlačítko v nástrojové liště (viz obrázek 5.1).

3. Zvolit jméno a umístění projektu (viz obrázek 5.1) a potvrdit tlačítkem OK.



Obrázek 5.1: Zakládání nového projektu.

4. Následuje výběr cílového hardwaru, možnost použití plného simulátoru procesoru nebo projekt s operačním systémem OSEK (viz obrázek 5.2). Po potvrzení je nový projekt založen (viz obrázek 5.3).



Obrázek 5.2: Výběr typu cílového zařízení.

5.1.3 Práce s projektem

Práce s projektem je obdobná jako u jiných vývojových nástrojů. Po založení nového projektu se vytvoří okno s projektem, které poskytuje tři pohledy na projekt (mezi jednotlivými pohledy se přepínáme klikáním na jednotlivé záložky v okně projektu, jak ukazuje obrázek 5.3):

1. soubory, které souvisejí s projektem,
2. práce linkeru a
3. cílový HW.

5.1.3.1 Soubory projektu

V tomto pohledu na projekt jsou vidět všechny soubory, které souvisejí s vytvořením nové aplikace. Se soubory projektu je možné provádět běžné činnosti jako je kopírování, mazání, přidávání souboru do projektu, vytváření nebo mazání složek atd. Projekt je označen jménem projektu a koncovkou *jméno.mcp*. V části *Source* jsou soubory se zdrojovým kódem programu, hlavičkové soubory a u projektu s operačním systémem OSEK zde nalezneme OIL soubor s definicí OSEK aplikace, který se vytvoří např. pomocí programu OSEK Builder. V části *Startup Code* je vždy zaváděcí soubor pro daný typ procesoru, který se při překladu připojí ke zdrojovému kódu programu a udává například adresu startu programu, inicializuje zásobník, paměť RAM a další. V části *OSEK* jsou soubory, které jsou potřebné při vytváření OSEK aplikace, obsahují definice jednotlivých funkcí a při překladu se připojují k programu. Dále v tomto okně nalezneme knihovny, *ini* soubor s nastavením kompilátoru, debugru a další (viz obrázek 5.3).

Kliknutím na požadovaný soubor se otevře okno, ve kterém se daný soubor edituje. V okně projektu jsou dále zobrazeny v nástrojové liště ikony, které slouží ke spuštění překladu, debugru a nastavení parametrů projektu (viz obrázek 5.3). Kliknutí pravým tlačítkem myši na soubor v okně projektu se zobrazí lokální nabídka, ze které je možené například daný soubor zkompilovat, zkontrolovat syntaxi a další.

5.1.3.2 Práce linkeru

Zde je vidět jak se bude kompilovat nebo linkovat výstupní soubor pro vybraný cílový HW. Jaké se použijí soubory a v jakém pořadí bude prováděna kompilace nebo linkování.

5.1.3.3 Cílový HW

Tento pohled na projekt dává informace o vybraném cílovém HW

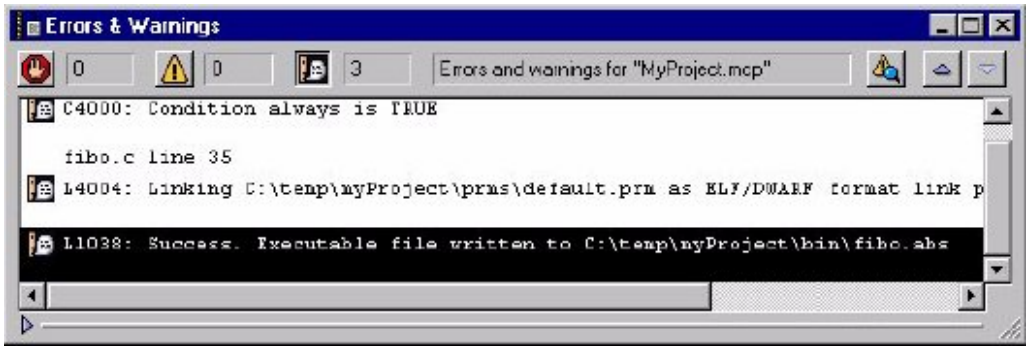
Obrázek 5.3: Ukázka práce s projektem.

V daném okamžiku může být spuštěno více projektů najednou. K těmto projektům se pak přistupuje zcela nezávisle. Pro každý projekt se tak může vytvořit individuální nastavení parametrů kompilátoru, linkeru, nástroje pro vypalování programu (*Burner*) do procesoru, nastavení parametrů cílového HW nebo simulátoru atd. Každý projekt může být vytvořen pro jeden nebo více cílových HW platforem, přičemž v rámci jednoho projektu mohou jednotlivé cílové HW platformy sdílet nějaké nebo všechny soubory. Při překladu je pak pouze nutné vybrat správný projekt a v něm typ cílového HW (viz obrázek 5.3).

5.1.3.4 Překlad projektu a vytvoření výstupního souboru

Následující postup ukáže jak se provádí překlad projektu.

1. Vybereme požadovaný projekt a v okně projektu klikneme na složku se soubory projektu, jak ukazuje obrázek 5.4.



Obrázek 5.5: Okno s výsledkem překladu.

5.1.3.5 Simulace a ladění programu

S vytvořeným absolutním souborem, který vznikl při překladu použitím příkazu *make*, je nyní možné provádět simulaci a ladění programu. V nástrojové liště v okně projektu klikneme na tlačítko *debugru* (viz obrázek 5.4). Tím dojde ke spuštění debugru (viz obrázek 5.6), ve kterém je možné provádět ladění programu. Pokud používáme podporovaný cílový HW, tak je možné provádět ladění přímo na procesoru v reálném čase. Pokud cílový HW není podporován, tak ladění probíhá v prostředí simulátoru, který plně simuluje daný procesor včetně možnosti připojení komponent k simulaci vstupů/výstupů (viz obrázek 5.6), které najdeme v hlavním menu *Component* $->$ *Open*.

Pro ladění programu debuger poskytuje funkce na krokování programu po jednom kroku, přeskočení funkce bez krokování, vyskočení z funkce, spuštění a zastavení programu. Dále dává pohled na zdrojový kód jak v jazyku C/C++, tak i v assembleru, poskytuje pohled na data v paměti, na obsah registrů procesoru a nakonec na obsah použitých proměnných, které chce uživatel sledovat, ty je možné přidat dvojím kliknutím na okno *data*. Pro lepší orientaci ve zdrojovém kódu programu nebo pro zastavení v určitém místě jsou k dispozici *breakpointy*, které se umísťují, tak že v místě kam chceme *breakpoint* umístit kliknem myší a pravým tlačítkem vyvoláme lokální nabídku, ze které vybereme *set breakpoint* a tím ho nastavíme. Stejným postupem se smaže nebo je možné vyvolat tabulku, která nám ukáže, kde jsou všechny *breakpointy* umístěné.

V každém okně, které nám debuger a simulátor nabízí, je možné vyvolat kliknutím pravého tlačítka myši lokální nabídku, ze které je možné si dané okno nastavit, zvolit jiný režim pohledu nebo změnit typ formátu pro zobrazení hodnot proměnných nebo adres atd. V hlavním menu *Component* $->$ *Set Target* je možné vybrat cílový HW, pokud aplikace byla psána pro více cílových HW najednou. Všechna nastavení pro odladění a od simulování aplikace, které se v simulátoru a debugru provedly, je možné uložit, takže při dalším spuštění simulátoru z daného projektu se automaticky toto nastavení provede.

5.2 OSEK Builder

OSEK Builder je nástroj, ve kterém se definuje aplikace, která se vytváří v real-time operačním systému OSEK. Zde se definují jednotlivé úlohy, komunikace mezi nimi, která úloha vlastní jaké události, alarmy, zprávy, systémové zdroje, čítače, obsluhy přerušení a další (viz obrázek 5.7).

Obrázek 5.6: Simulátor a debuger.

Obrázek 5.7: OSEK Builder.

Po vytvoření definice OSEK aplikace se provede *verifikace standardu*, tzn. provede se kontrola, zda takto definovaná aplikace vyhovuje zvolené implementaci operačního systému. Tuto kontrolu nalezneme v hlavním menu *Project* − > *Verify Standard* . Po uložení této definice OSEK aplikace vzniká soubor *.OIL, což je soubor, který obsahuje definici celé aplikace v implementačním jazyce operačního systému OSEK, tzv. OIL (*OSEK Implementation Language*).

5.2.1 Struktura hlavního souboru OSEK aplikace *main.c*

Po založení nového projektu pro vytvoření OSEK aplikace se mimo jiné také založí soubor *main.c*, do kterého se píše vlastní kód aplikace, tedy vlastní těla jednotlivých úloh. Tento soubor v sobě již obsahuje předdefinovanou strukturu, tzn. že v sobě již obsahuje potřebné náležitosti pro napojení na operační systém.

Na začátku tohoto souboru se nacházejí odkazy na hlavičkové soubory, v kterých jsou zadefinovány vlastnosti operačního systému, dále pak zde nalezneme odkaz na volání API nebo odkaz na hlavičkové soubory s konfigurací aplikace (viz obrázek 5.8). Dále následuje deklarace úloh, alarmů, událostí, čítačů a systémových zdrojů, které jsme použili při definici aplikace ve vývojovém prostředí OSEK Builder (viz kapitola 5.2). Nyní následuje hlavní funkce *main.c*, která zajišťuje start operačního systému pomocí služby *StartOS* (viz kapitola 4.2.11), která má jako parametr aplikační mód *appmode*. Dále se již nachází zdrojový kód jednotlivých úloh.

```
#else   /* !defined(APPTYPESH)  */
#include    APPTYPESH                          /* header file for sample typedefs, defines etc. */
#endif  /* !defined(APPTYPESH)  */

/************************* System Include Files ******************************/
#if !defined(OSPROPH)
#include    "osprop.h"                         /* OS Properties */
#else   /* !defined(OSPROPH)    */
#include    OSPROPH                            /* OS Properties */
#endif  /* !defined(OSPROPH)    */

#include <osapi.h>                             /* API calls */

#if !defined(OSCFGH)
#include    "cfg.h"                            /* Application configuration header file */
#else   /* !defined(OSCFGH) */
#include    OSCFGH                             /* Application configuration header file */
#endif  /* !defined(OSCFGH) */

/************************* Project Include Files ****************************/

/************************* OSEK Declarations ********************************/
DeclareTask( );

DeclareAlarm( );

DeclareEvent( );

DeclareResource( );

DeclareCounter( );

/*****************************************************************************/
void
main(void)
{
    StartOS( appmode );      /* jump to OSEKturbo OS 2.1 startup */
}

/*****************************************************************************/
TASK ()
{

   TerminateTask();
}
```

Zde se provádí deklarace

Zde se spouští operační systém

Zde se píše uživatelský kód dané úlohy

Obrázek 5.8: Struktura hlavního souboru *main.c*.

5.3 Vytvoření OSEK aplikace

Zde je uveden kompletní postup pro vytvoření aplikace s operačním systémem OSEK. Postup bude ukázán na vývojovém prostředí CodeWarrior a OSEK Builder od společnosti Metrowerks, kterou vlastní společnost Motorola a tudíž implementace operačního systému je právě pro procesory Motorola řady HC08 a HC12. Postup je následující:

1. Ve vývojovém prostředí CodeWarrior založíme projekt (viz kapitola 5.1.2) s operačním systémem OSEK pro daný typ procesoru.

2. V okně projektu vybereme soubor *osdef.oil*, na který když dvakrát klikneme myší, se spustí program OSEK Builder, ve kterém se provede definice OSEK aplikace.

3. V programu OSEK Builder se provede definice OSEK aplikace, tzn. vyberou se příslušné objekty operačního systému pro založení úloh, definice alarmů, zpráv, událostí a dalších a nastaví se u nich požadované atributy.

4. Po té co vytvoříme popis OSEK aplikace v OIL jazyce, provedeme kontrolu zda popis odpovídá zvolené implentaci operačního systému.

5. Nyní se vrátíme do prostředí CodeWarrior, kde soubor *osdef.oil* zkompilujeme. Tím se v okně projektu vytvoří složka *OSEK_Gen*, ve které se nacházejí konfigurační soubory námi vytvořené OSEK aplikace v jazyce C.

6. Nyní otevřeme z okna projektu hlavní soubor OSEK aplikace *main.c*, do kterého nejprve zadeklarujeme námi definované úlohy, události, alarmy, čítače, systémové zdroje a po té začneme psát zdrojový kód jednotlivým úlohám (viz obrázek 5.2.1 struktura hlavního souboru main.c).

7. Po dokončení následuje překlad a linkování celého projektu příkazem *make*.

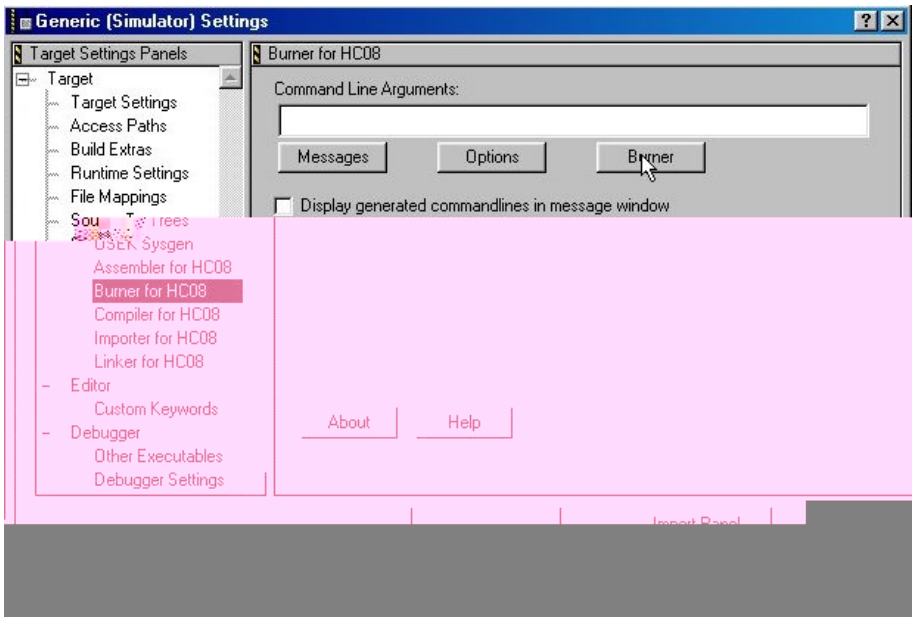
8. Nyní provedeme debugování a simulaci aplikace.

9. Nakonec vytvoříme výstupní spustitelný soubor (viz kapitola 5.4) pro daný typ procesoru v požadovaném formátu: *.HEX, *.s19 nebo Binární soubor. Tento soubor následně vypálíme do procesoru.

5.4 Vytvoření výstupního souboru a nahrání do procesoru Motorola HC08

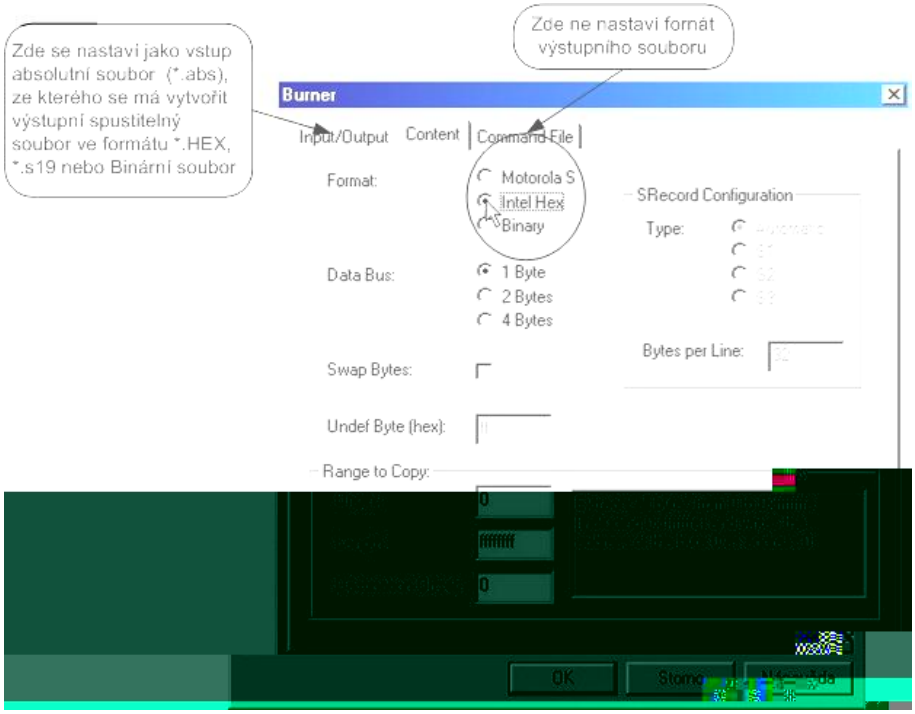
5.4.1 Vytvoření výstupního souboru

Pro vytvoření výstupního spustitelného souboru je ve vývojovém prostředí CodeWarrior utilita, tzv. *burner*. Tu spustíme z okna projektu (viz obrázek 5.4) kliknutím na tlačítko *setting* v nástrojové liště, dále kliknem na položku *burner for HC08* a následně na tlačítko *burner* (viz obrázek 5.9).



Obrázek 5.9: Utilita pro vytvoření výstupního souboru (*burner*).

Tím se spustí okno (viz obrázek 5.10), kde se provede nastavení. Nejprve klikneme na záložku *Input/Output*, kde nastavíme, ze kterého souboru se má výstupní soubor vytvořit. V poli *Input* nastavíme jako vstup absolutní soubor (*.abs), který jsme vytvořili při překladu projektu příkazem *make*. Ten se nachází v adresáři projektu na této cestě: $\backslash projekt \backslash bin \backslash * .abs$. Do pole *Output* napíšeme cestu, kam se má umístit výstupní soubor. V záložce *Content* vybereme požadovaný formát výstupního souboru. Na výběr jsou formáty: Motorola S (ten vytvoří soubor *.s19), Intel HEX nebo Binární soubor. Formát výstupního souboru volíme podle toho, jaký používáme software na vypalování našeho programu do procesoru. Po výběru formátu výstupního souboru se opět vrátíme do záložky *Input/output* a klikneme na tlačítko *Execute* a tím dojde k vytvoření výstupního souboru ve zvoleném formátu. Nyní můžeme přistoupit k vypálení programu do procesoru.



Obrázek 5.10: Nastavení formátu výstupního souboru.

5.4.2 Vypálení programu do procesoru

Postupy jak vypálit program do procesoru mohou být různé. Pokud vlastníme výstupní hardware, který je podporován vývojovým prostředím CodeWarrior, tak můžeme pro vypálení programu do procesoru použít přímo utilitu *burner*, kde se pouze nastaví k jakému sériovému portu počítače máme cílový hardware připojen, případně jakou rychlostí je schopen komunikovat.

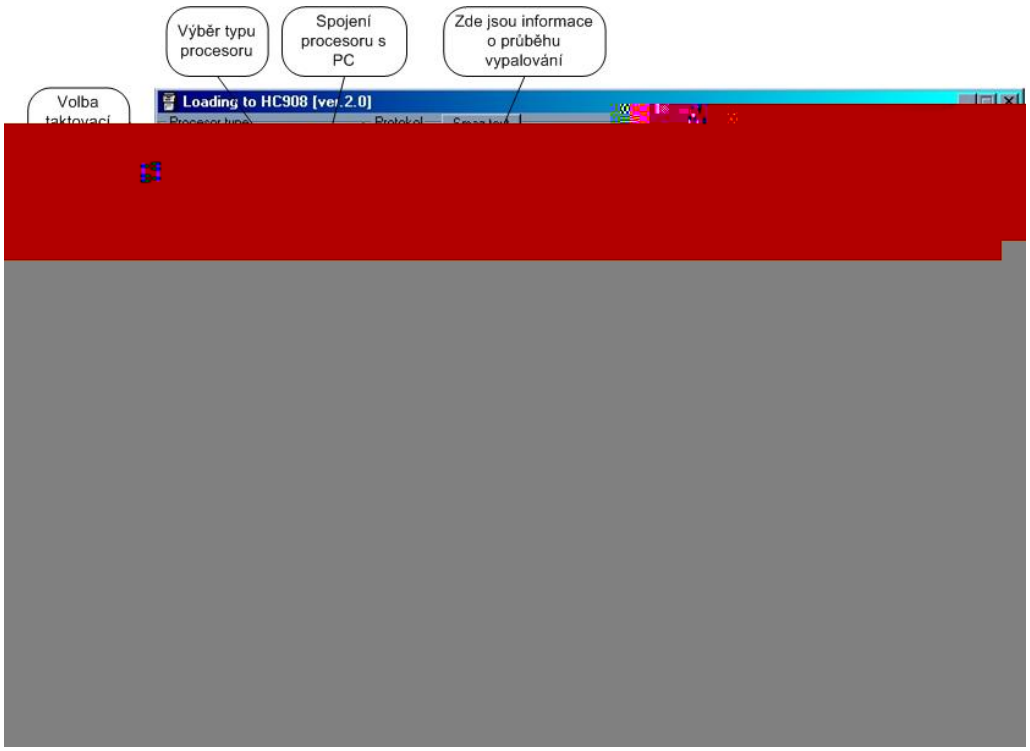
Jiný způsob je takový, že máme vlastní programátor a k němu používáme vlastní software pro vypálení programu do procesoru. To byl můj případ. Jako cílový hardware jsem používal desku s procesorem Motorola řady HC08, kterou jsem získal od společnosti UniControls a.s. Pro vypálení programu do procesoru bylo tedy nutné vyrobit modul pro komunikaci procesoru s PC pomocí sériového rozhraní. Schéma tohoto modulu, desku plošného spoje, součástky a software pro vypálení programu do procesoru rovněž poskytla společnost UniControls a.s. Já jsem tento modul poté vyrobil. Problém však byl ve vypalovacím softwaru, protože ten obsahoval několik chyb. Mezi největší patřila ta, že pokud jsem se snažil do procesoru vypálit aplikaci s operačním systémem OSEK, tak se to nikdy nezdařilo. Z tohoto důvodu jsem tedy nemohl takovou aplikaci vyzkoušet na reálném procesoru, pouze jsem ji mohl od simulovat v simulatoru vývojového prostředí CodeWarrior. Pokud jsem vytvořil aplikaci, která neobsahovala operační systém, tak vše proběhlo bez problémů a program se povedl do procesoru nahrát a následně spustit.

5.4.3 Postup jak výstupní soubor vypálit do procesoru

Zde je uveden postup jak vypálit vytvořený program do procesoru Motorola řady HC08 s programátorem a vypalovacím softwarem od společnosti UniControls a.s.:

1. Spustíme program *Loading to HC908 v.2.0* (viz obrázek 5.11).
2. V programu nastavíme správnou taktovací frekvenci procesoru, vybereme správný sériový port, pomocí kterého je procesor připojen přes programátor k PC.
3. Klikneme na tlačítko *Spojit se*, tím dojde ke spojení procesoru s PC.
4. Nastavíme cestu, kde se nachází náš program ve správném formátu. Ten může být jak ve formátu s19, tak i ve formátu HEX.
5. Nyní již můžeme přistoupit k samotného vypálení. To můžeme provést nejjednodušeji stiskem tlačítka *Tradičně* (tím se provede automatické nahrání programu do procesoru), které se nalézá v levém dolním rohu programu (viz obrázek 5.11). Po celou dobu náhrávání programu do procesoru program informuje zda správně načetl náš program, co právě provádí, kam co nahrává a nakonec provede kontrolu, zda je program vypálen správně.
6. Proběhlo-li všechno v pořádku, můžeme program spustit stiskem tlačítka *Go*.

Poznámka: Program se dá také vypalovat po jednotlivých krocích, tzn. postupným stiskem tlačítek *Start* $->$ *Obnov* $->$ *Loader* $->$ *Maž* $->$ *Zápis* $->$ *Check* $->$ *Go*, které provedou spojení s procesorem, ověření ochranného hesla, smazání pamětí, zápis programu, kontrolu zápisu a nakonec spuštění programu.



Obrázek 5.11: Software pro vypálení programu do procesoru.

Kapitola 6

Vzorové příklady OSEK aplikací

Na následujících stranách budou uvedeny vzorové příklady popisující možné způsoby použití služeb operačního systému OSEK. Postupně zde budou uvedeny příklady využívající služeb správy úloh, událostí, čítačů, správy zdrojů, zpráv a alarmů.

6.1 Příklad 1: Správa úloh

Tento příklad ukazuje použití služeb operačního systému pro správu úloh. Příklad obsahuje tři úlohy *TASK_A*, *TASK_B*, *TASK_C*, které využívají služeb operačního systému pro vzájemnou koordinaci.

6.1.1 Použité služby operačního systému

Následují text obsahuje popis služeb operačního systému, které byly použity v tomto příkladě.

`DeclareTask(`⟨***jméno úlohy***⟩`)`: *DeclareTask* slouží jako vnější deklarace úlohy, touto deklarací se sděluje deklarování funkce úlohy *TASK(⟨jméno úlohy⟩)*.

`ActivateTask(`***TaskType*** ⟨***TaskID***⟩`)`: tato služba zajišťuje přechod úlohy ⟨*TaskID*⟩ ze stavu *suspended* do stavu *ready*. Pokud úloha není ve stavu *suspended*, tak je tato služba ignorována.

`Schedule( )` : tato služba zajišťuje přeplánování, tzn. že pokud existuje úloha s vyšší prioritou ve stavu *ready*, tak úloha, která tuto službu volala přechází do stavu *ready* a úloha s vyšší prioritou přechází do stavu *running*. Jinak pokračuje volání úlohy.

`GetTaskID(`***TaskRefType*** ⟨***TaskIDRef***⟩`)`: tato služba vrací odkaz na úlohu, která je pravě ve stavu *running*. Pokud úloha právě neběží, tak služba vrací konstantu INVALID_TASK.

`ChainTask(`***TaskType*** ⟨***TaskID***⟩`)`: tato služba způsobuje ukončení úlohy, která tuto funkci volala a zároveň zajišťuje přechod následující úlohy ⟨TaskID⟩ ze stavu *susspended* do stavu *ready*. Volání této služby zajišťuje, že následují úloha bude spuštěna poté co bude ukončena úloha, která tuto službu volala.

GetTaskState(*TaskType* ⟨*TaskID*⟩, *TaskStateRefType* ⟨*StateRef*⟩): tato služba vrací stav v jakém se nachází úloha ⟨TaskID⟩ (*running*, *ready*, *waiting*, *susspended*) v okamžiku volání služby *GetTaskState*.

TerminateTask( ): tato služba způsobuje ukončení úlohy, která tuto službu volala a ta přechází ze stavu *running* do stavu *ready*.

6.1.2 Popis příkladu

Příklad obsahuje čtyři úlohy (viz obrázek 6.1) *INIT_TASK* s prioritou 5, *TASK_A* s prioritou 3, *TASK_B* s prioritou 2 a *TASK_C* s prioritou 1.

Úloha *INIT_TASK* má nejvyšší prioritu, tak je po zavedení systému spuštěna jako první a provede to, že úlohu *TASK_A* uvede do stavu *ready* a provede inicializaci systémového čítače *SYSTEMTIMER*. Poté přejde do stavu *suspended*, protože již není víckrát aktivována jinou úlohou, tak v tomto stavu zůstává natrvalo.

Nyní se spustí úloha, která je ve stavu *ready* a má nejvyšší prioritu. Takovouto úlohou je úloha *TASK_A*. Tato úloha nyní aktivuje úlohy *TASK_B* a TASK_C a uvádí je do stavu *ready*. nakonec je *TASK_A* zrušena a přechází do stavu *suspended*.

V tomto okamžiku máme *TASK_B* a úlohu *TASK_C* ve stavu *ready*, ale protože úloha *TASK_B* má vyšší prioritu, tak přechází do stavu *running*. Následně zjišťuje v jakém stavu se nachází úloha *TASK_C*, zavoláním služby *GetTaskState( )*. Nachází-li se ve stavu *waiting*, tak nastavuje událost *TASK_C_EVENT*. Jestliže se nachází ve stavu *suspended*, tak je po zavolání služby *ChainTask( )* (která zruší úlohu, která ji volala a aktivuje následující úlohu) uvedena do stavu *ready*. Úloha *TASK_B* je zrušena a tím přechází do stavu *suspended*.

Nyní se ve stavu *ready* nachází pouze úloha *TASK_C*, tudíž přechází do stavu *running*. Tato úloha testuje opět pomocí služby *GetTaskState* v jakých stavech se nacházejí zbylé úlohy a podle toho je aktivuje. Na jejím konci je úloha *TASK_C* zrušena a úlohy *TASK_A* a *TASK_B* jsou ve stavu *ready*, ale protože úloha *TASK_A* má vyšší prioritu, tak je spuštěna ta a celá aplikace se opakuje.

Obrázek 6.1: Popis příkladu 1.

6.2 Příklad 2: Použití služeb pro správu systémových zdrojů

Tento příklad demonstruje použití služeb pro správu sdílených systémových zdrojů.

6.2.1 Použité služby operačního systému

Následující text uvádí popis použitých systémových služeb pro správu systémových zdrojů použitých v tomto příkladě.

DeclareResource(⟨***jméno systémového zdroje***⟩)

DeclareTask(⟨***jméno úlohy***⟩): *DeclareTask* slouží jako vnější deklarace úlohy, touto deklarací se sděluje deklarování funkce úlohy *TASK(⟨jméno úlohy⟩)*.

GetResource(***ResourceType*** ⟨***ResID***⟩): tato služba změní hodnotu aktuální priority na hodnotu maximální dostupné priority pro správu zdrojů. Služba *GetResource* slouží jako vstup do kritické sekce kódu a bloku spouštěného některou z úloh nebo přerušovací rutinou, která může dostat systémový zdroj ⟨*ResID*⟩. Kritická sekce musí být vždy ukončena voláním funkce *ReleaseResource* uvnitř té samé úlohy nebo obsluhy přerušení.

ReleaseResource(***ResourceType*** ⟨***ResID***⟩): volání této služby zajišťuje opuštění kritické sekce kódu, která je přidělena systémovému zdroji, který je odkazován pomocí ⟨ResID⟩. Služba *ReleaseResource* se volá jako protějšek služby *GetResource*. Tato služba vrací úlohu nebo obsluhu přerušení do úrovně v jaké byla před voláním odpovídající služby *GetResource*.

ActivateTask(***TaskType*** ⟨***TaskID***⟩): tato služba zajišťuje přechod úlohy ⟨*TaskID*⟩ ze stavu *suspended* do stavu *ready*. Pokud úloha není ve stavu *suspended*, tak je tato služba ignorována.

GetTaskState(***TaskType*** ⟨***TaskID***⟩, ***TaskStateRefType*** ⟨***StateRef***⟩): tato služba vrací stav v jakém se nachází úloha ⟨TaskID⟩ (*running*, *ready*, *waiting*, *susspended*) v okamžiku volání služby *GetTaskState*.

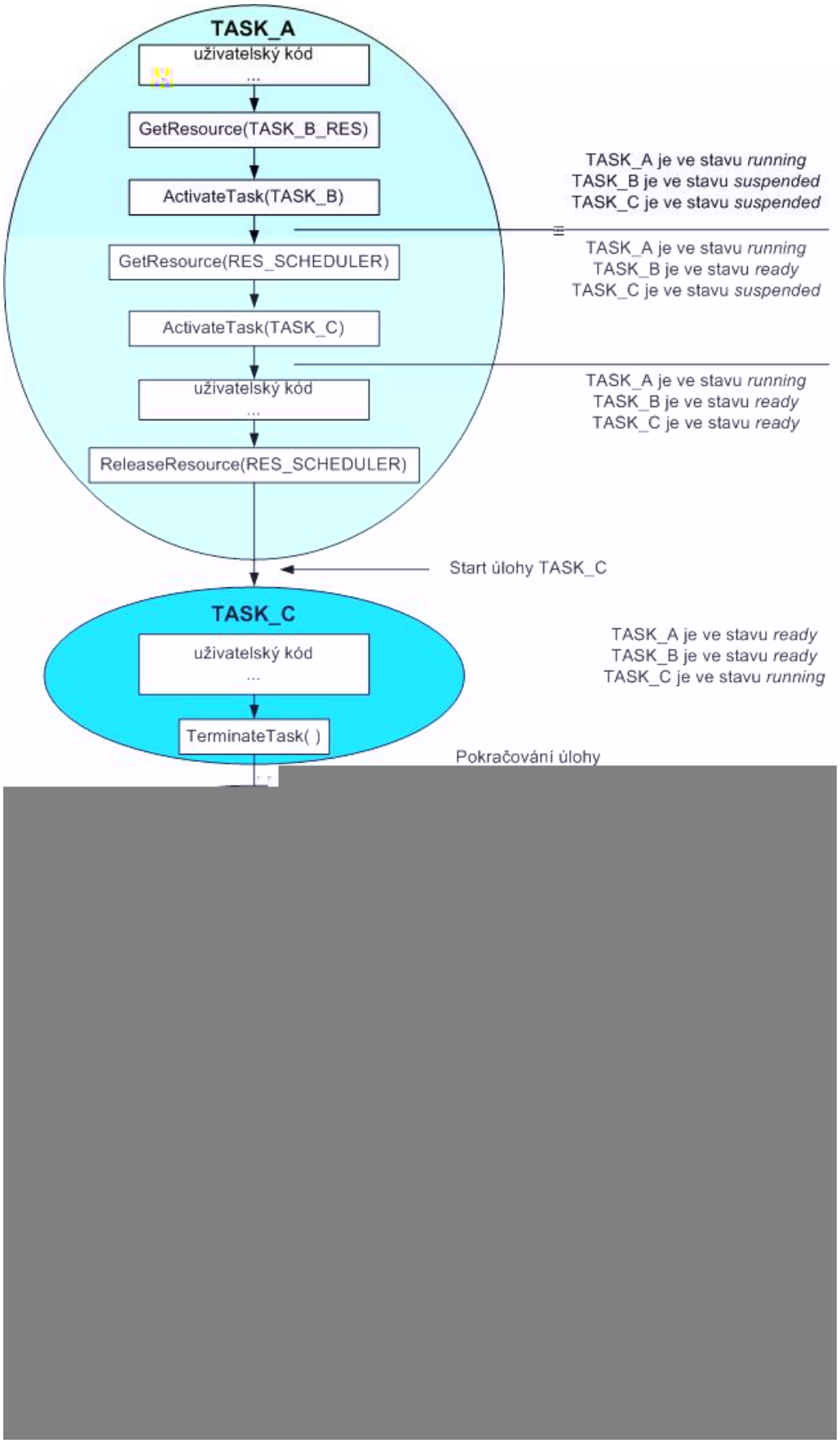
TerminateTask( ): tato služba způsobuje ukončení úlohy, která tuto službu volala a ta přechází ze stavu *running* do stavu *ready*.

6.2.2 Popis příkladu

Příklad obsahuje čtyři úlohy *INIT_TASK* s prioritou 5, *TASK_A* s prioritou 1, *TASK_B* prioritou 2 a *TASK_C* s prioritou 3. Přičemž úloha *TASK_A* sdílí systémový zdroj *TASK_B_RES* s úlohou *TASK_B*. Úloha také používá systémový zdroj *RES_SCHEDULER*, což je vlastně plánovač, který je chápán jako systémový zdroj. Použití tohoto systémového zdroje je nutné nastavit ve vývojovém prostředí OSEK aplikace, tj. v programu OSEK Builder jako jeden z atributů u systémového objektu *OS*.

Úloha *INIT_TASK* má nejvyšší prioritu, a proto bude spuštěna jako první a provede inicializaci systémového časovače *SYSTEMTIMER* a je ukončena. Tím se dostává do stavu *suspended*, ve kterém zůstane do konce aplikace, protože ji jiná úloha neaktivuje. Aplikace dále pokračuje ve vykonávání úlohy *TASK_A* (viz obrázek 6.2), protože je ve

stavu *ready* a má momentálně nejvyšší prioritu. Ta provádí svůj uživatelský kód. Následně obsazuje systémový zdroj *TASK_B_RES*, aby zabránila startu úlohy *TASK_B* voláním služby *GetResource*. Pak aktivuje úlohu *TASK_B* a uvede ji do stavu *ready*. Nyní obsazuje plánovač voláním funkce *GetResource(RES_SCHEDULER)*, aby nedošlo k přeplánování. Aktivuje úlohu *TASK_C*, tím ji uvede do stavu *ready*. Dále následuje opět uživatelský kód. Poté opouští systémový zdroj *RES_SCHEDULER* a v tomto okamžiku dochází ke spuštění úlohy *TASK_C* a k přerušení vykonávání úlohy *TASK_A*. Po dokončení úlohy *TASK_C* opět pokračuje ve svém vykonávání úloha *TASK_A*. Nyní úloha *TASK_A* opouští systémový zdroj *TASK_B_RES*, čímž dochází ke spuštění úlohy *TASK_B* a tedy k přerušení úlohy *TASK_A*. Po dokončení úlohy *TASK_B* dochází opět ke spuštění úlohy *TASK_A* a k jejímu dokončení. Celý vývoj této aplikace je zachycen na obrázku 6.2.



Obrázek 6.2: Popis příkladu 2.

6.3 Příklad 3: Události

V tomto příkladě bude ukázáno použití události.

6.3.1 Použité služby operačního systému

Následující text uvádí popis použi-tých systémových služeb pro správu událostí použitých v tomto příkladě. Navíc jsou zde také použity služby pro poslání a příjem zprávy a cyklický alarm pro aktivaci úlohy.

DeclareEvent(⟨***jméno události***⟩)

DeclareTask(⟨***jméno úlohy***⟩): *DeclareTask* slouží jako vnější deklarace úlohy, touto deklarací se sděluje deklarování funkce úlohy *TASK(⟨jméno úlohy⟩)*.

DeclareResource(⟨***jméno systémového zdroje***⟩)

SetEvent(***TaskType⟨TaskID⟩, EventMaskType⟨Mask⟩***): tato služba slouží k nastavení jedné nebo několika událostí požadované úlohy podle masky události. Pokud úloha čeká alespoň na jednu ze specifikovaných událostí, pak je uvedena do stavu *ready*. Události nespecifikované maskou zůstanou nezměněny. Na množinu událostí může být odkazována pouze rozšířená úloha, která není ve stavu *suspended*.

ClearEvent(***EventMaskType⟨Mask⟩***): Úloha, která volá tuto službu definuje, která událost bude smazána.

GetEvent(***TaskType⟨TaskID⟩, EventMaskType⟨Event⟩***): Maska události, která je uvedena ve volání, je vyplněna podle stavu události požadované úlohy. Je vrácen aktuální stav události, ale ne maska události, na kterou úloha čeká.

WaitEvent(***EventMaskType⟨Mask⟩***): úloha, která volá tuto službu je uvedena do stavu *waiting* dokud není nastavena alespoň jedna událost specifikovaná maskou. Úloha zůstává ve stavu *running* pokud dojde k nastavení události v okamžiku volání služby.

GetResource(***ResourceType ⟨ResID⟩***): tato služba změní hodnotu aktuální priority na hodnotu maximální dostupné priority pro správu zdrojů. Služba *GetResource* slouží jako vstup do kritické sekce kódu a bloku spouštěného některou z úloh nebo přerušovací rutinou, která může dostat systémový zdroj ⟨*ResID*⟩. Kritická sekce musí být vždy ukončena voláním funkce *ReleaseResource* uvnitř té samé úlohy nebo obsluhy přerušení.

ReleaseResource(***ResourceType ⟨ResID⟩***): volání této služby zajišťuje opuštění kritické sekce kódu, která je přidělena systémovému zdroji, který je odkazován pomocí ⟨*ResID*⟩. Služba *ReleaseResource* se volá jako protějšek služby *GetResource*. Tato služba vrací úlohu nebo obsluhu přerušení do úrovně v jaké byla před voláním odpovídající služby *GetResource*.

SendMessage(***SymbolicName*** ⟨***Message***⟩, ***AccessNameRef*** ⟨***Data***⟩): *Nefrontová WithCopy zpráva:* tato služba aktualizuje objekt zpráva ⟨*Message*⟩ s daty danými ⟨*Data*⟩ a žádosti přenosu objektu zprávy příjemci. Objekt zpráva je uzamknut během aktualizace a během přenosu dat nižší komunikační vrstvě. Není aktualizován pokud již je uzamknut. *Nefrontová WihtoutCopy zpráva:* tato služba vyžaduje přenos už aktualizovaného objektu zprávy pro příjem. Tato služba také aktualizuje stav informace v objektu zpráva.

ReceiveMessage(***SymbolicName*** ⟨***Message***⟩, ***AccessNameRef*** ⟨***Data***⟩): *Nefrontová WithCopy zpráva:* tato služba doručuje data zprávy, které jsou spojené s objektem zpráva ⟨*Message*⟩. Objekt zpráva je během příjmu dat uzamknut. Tato služba také aktualizuje stav informace v objektu zpráva. *Nefrontová WihtoutCopy zpráva:* služba s tímto parametrem vrací status pouze od aplikace, která může přistupovat přímo k objektu zpráva.

DeclareAlarm(⟨***jméno alarmu***⟩)

SetRelAlarm(***AlarmType*** ⟨***AlarmID***⟩, ***TickType*** ⟨***Increment***⟩, ***TickType*** ⟨***Cycle***⟩): systémová služba okupuje alarm ⟨*AlarmID*⟩. Po ⟨*inkrementaci*⟩ čítače dojde k jeho vypršení, tím dojde k aktivaci úlohy nebo nastavení události (pouze u rozšířené události), která je k tomuto alarmu ⟨*AlarmID*⟩ přiřazena. Pokud *Cycle* se nerovná 0, tak je alarm nastaven ihned po vypršení relativní hodnoty *Cycle*. Jinak je alarm spouštěn pouze jednou. Jestliže je relativní hodnota *Increment* rovna 0, tak alarm vyprší ihned a úloha, ke které je nastaven přejde do stavu *ready* předtím, něž se systémová služba vrátí do volané úlohy nebo ISR.

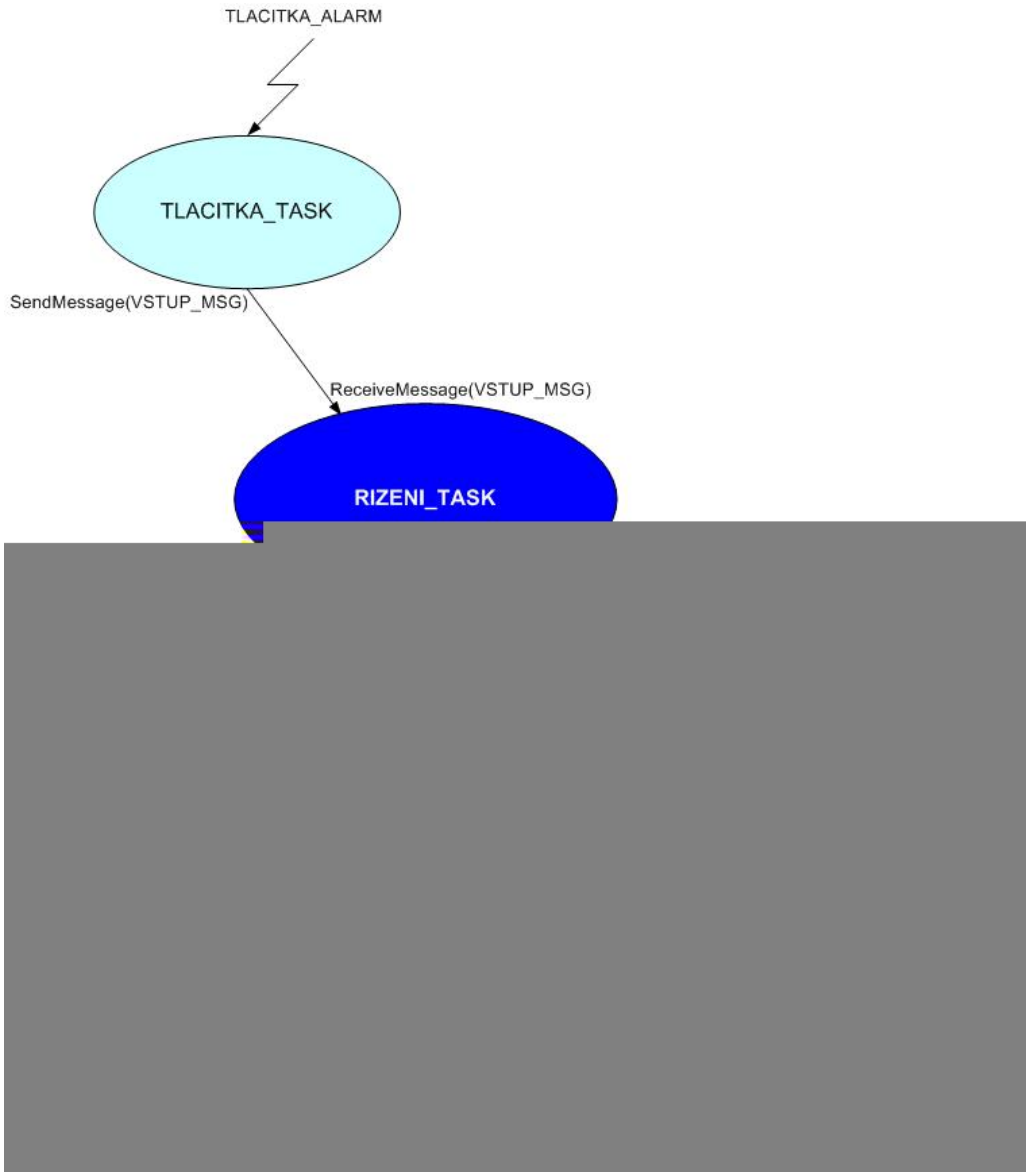
TerminateTask( ): tato služba způsobuje ukončení úlohy, která tuto službu volala a ta přechází ze stavu *running* do stavu *ready*.

6.3.2 Popis příkladu

Tato aplikace obsahuje pět úloh *INIT_TASK* s prioritou 50, *TLACITKA_TASK* s prioritou 0, *RIZENI_TASK* s prioritou 40, *VAGON_TASK* s prioritou 30 a *LED_TASK* s prioritou 10 (viz obrázek 6.3). Úloha *VAGON_TASK* vlastní tři události: *MOTOR_L_EVENT*, *MOTOR_R_EVENT* a *STOP_EVENT*, tzn. že se jedná o rozšířenou úlohu.
Úloha *LED_TASK* vlastní dvě události: *SEN_L_EVENT* a *SEN_R_EVENT*, tzn že se jedná také o rozšířenou úlohu.

Jako první aplikace spustí úlohu *INIT_TASK*, protože je stavu *ready* a má nejvyšší prioritu. Provede aktivaci úloh *VAGON_TASK* a *LED_TASK*, tzn. že je uvede do stavu *ready*. Dále provede inicializaci systémového časovače *SYSTEMTIMER*, na který je napojen alarm *TLACITKA_ALARM*. Úloha *TLACITKA_TASK* je cyklicky spouštěna pomocí alarmu *TLACITKA_ALARM*. Jakmile dojde ke stiknutí tlačítka *run* úloha *TLACITKA_TASK* vysílá zprávu *VSTUP_MSG* úloze *RIZENI_TASK*. Ta ji přijímá, zjišťuje že bylo stisknuto tlačítko *run*. Vyhodnotí, u jaké zarážky se nacházel vozík naposledy, podle toho nastaví události, které určují jakým směrem se má pohybovat vozík a spouštějí signalizaci. Vyhodnotí-li úloha *RIZENI_TASK*, že bylo stisknuto tlačítko *stop*, nastaví rovněž příslušnou událost. Úlohy *VAGON_TASK* a *LED_TASK* tedy čekají ve stavech *waiting* na to, až nastane daná událost, kterou daná úloha vlastní. Tím vždy aktivuje příslušnou úlohu a ta se provede. Stisknutím tlačítka *run* se vozík pohybuje od jedné zarážky ke druhé.

Směr, kterým se pohybuje, je rovněž možné sledovat jako šipky na display. Po stisknutí tlačítka *stop* se vozík zastaví a display zhasne.



Obrázek 6.3: Popis příkladu 3.

6.4 Příklad 4: Čítače

V tomto příkladě bude ukázáno použití čítačů.

6.4.1 Použité služby operačního systému

Následující text uvádí popis použitých systémových služeb použitých v tomto příkladě.

DeclareCounter(⟨***jméno čítače***⟩)

InitCounter(***CtrRefType*** ⟨***CounterID***⟩, ***TickType*** ⟨***Ticks***⟩): Nastavuje počáteční hodnotu čítače v ⟨Ticks⟩. Po tomto nastavení čítač zvýší svoji hodnotu voláním služby *CounterTrigger*. Pokud zde běží připojené alarmy, tak jejich stav zůstane nezměněn, ale čas vypršení nastane neurčitě.

CounterTrigger(***CtrRefType*** ⟨***CounterID***⟩): tato služba zvýší současnou hodnotu čítače. Pokud se čítač dostane na maximální dovolenou hodnotu, kterou má nastavenou při definici aplikace v OSEK Builderu ve svém atributu, tak je resetován na 0. Je-li alarm připojen k čítači, tak systém kontroluje, zda vypršely po správném počtu tiků a zda provedl požadovanou operaci (aktivace úlohy nebo nastavení události).

GetCounterValue(***CtrRefType*** ⟨***CounterID***⟩, ***TickRefType*** ⟨***TicksRef***⟩): služba poskytuje aktuální hodnotu čítače ⟨*CounterID*⟩ v ticích a uloží ji do proměnné odkazované pomocí ⟨*TicksRef*⟩.

DeclareAlarm(⟨***jméno alarmu***⟩)

SetRelAlarm(***AlarmType*** ⟨***AlarmID***⟩, ***TickType*** ⟨***Increment***⟩, ***TickType*** ⟨***Cycle***⟩): systémová služba okupuje alarm ⟨*AlarmID*⟩. Po ⟨*inkrementaci*⟩ čítače dojde k jeho vypršení, tím dojde k aktivaci úlohy nebo nastavení události (pouze u rozšířené události), která je k tomuto alarmu ⟨*AlarmID*⟩ přiřazena. Pokud *Cycle* se nerovná 0, tak je alarm nastaven ihned po vypršení relativní hodnoty *Cycle*. Jinak je alarm spouštěn pouze jednou. Jestliže je relativní hodnota *Increment* rovna 0, tak alarm vyprší ihned a úloha, ke které je nastaven přejde do stavu *ready* předtím, něž to systémová služba vrátí volané úloze nebo ISR.

DeclareTask(⟨***jméno úlohy***⟩): *DeclareTask* slouží jako vnější deklarace úlohy, touto deklarací se sděluje deklarování funkce úlohy *TASK(⟨jméno úlohy⟩)*.

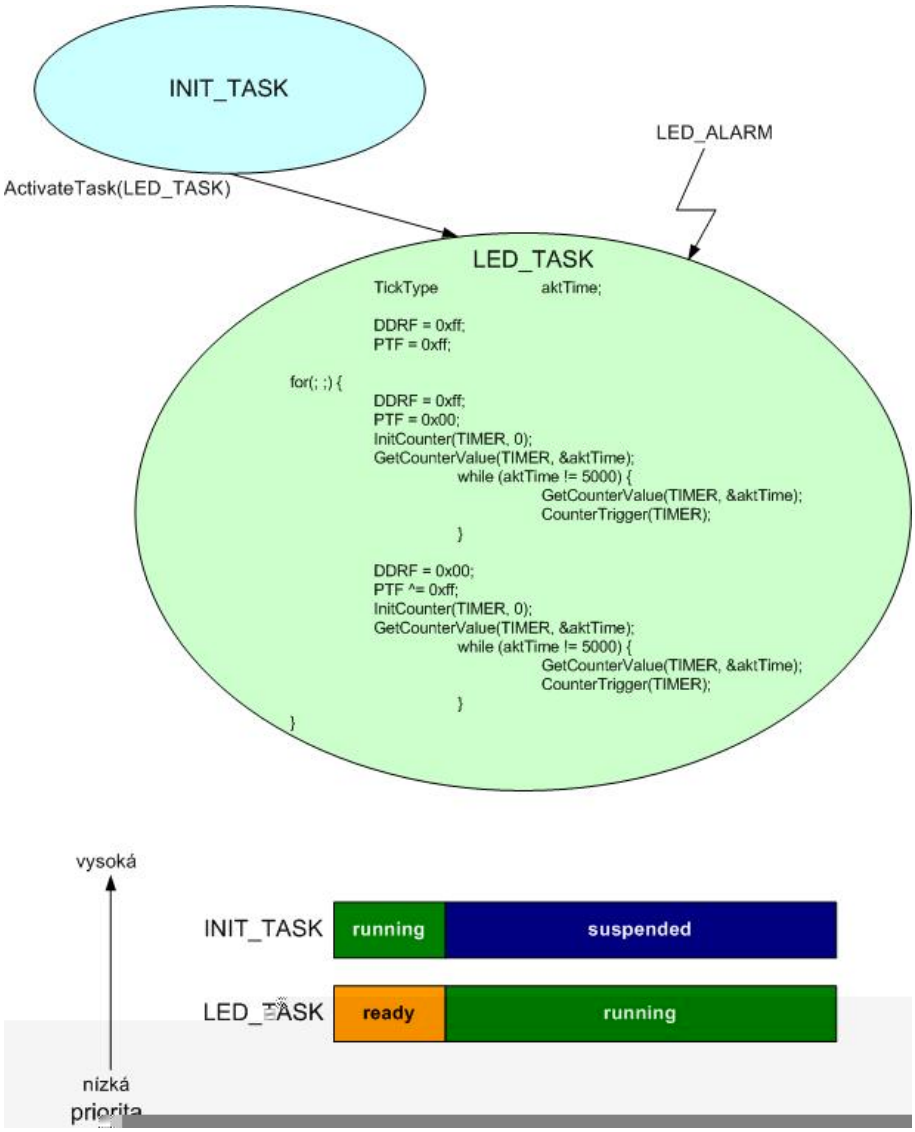
ActivateTask(***TaskType*** ⟨***TaskID***⟩): tato služba zajišťuje přechod úlohy ⟨*TaskID*⟩ ze stavu *suspended* do stavu *ready*. Pokud úloha není ve stavu *suspended*, tak je tato služba ignorována.

TerminateTask( ): tato služba způsobuje ukončení úlohy, která tuto službu volala a ta přechází ze stavu *running* do stavu *ready*.

6.4.2 Popis příkladu

Tato aplikace obsahuje dvě úlohy *INIT_TASK* s prioritou 5 a úlohu *LED_TASK*, která je cyklicky spouštěna alarmem *LED_ALARM* (viz obrázek 6.4).

Aplikace nejprve spouští úlohu *INIT_TASK*, protože je ve stavu *ready* má nejvyšší prioritu. Ta aktivuje úlohu *LED_TASK* a uvede ji do stavu *ready*, dale ještě nastaví cyklický alarm pro spouštění opakované úlohy *LED_TASK*. Poté je tato úloha ukončena a tím přejde do stavu *suspended* a v něm již zůstane, protože ji jiná úloha již neaktivuje. Poté je spuštěna úloha *LED_TASK*, protože je jako jako jediná ve stavu *ready*. Tato úloha provádí to, že na mění jeden z bitů na I/O portu F procesoru, na kterém je zapojena LED dioda, která v nastaveném čase bliká.



Obrázek 6.4: Popis příkladu 4.

6.5 Příklad 5: Zprávy

V tomto příkladě bude ukázána komunikace mezi úlohami pomocí zpráv.

6.5.1 Použité služby operačního systému

Následují text obsahuje popis služeb operačního systému, které byly použity v tomto příkladě.

DeclareTask(⟨***jméno úlohy***⟩): *DeclareTask* slouží jako vnější deklarace úlohy, touto deklarací se sděluje deklarování funkce úlohy *TASK(⟨jméno úlohy⟩)*.

TerminateTask( ): tato služba způsobuje ukončení úlohy, která tuto službu volala a ta přechází ze stavu *running* do stavu *ready*.

ActivateTask(***TaskType*** ⟨***TaskID***⟩): tato služba zajišťuje přechod úlohy ⟨*TaskID*⟩ ze stavu *suspended* do stavu *ready*. Pokud úloha není ve stavu *suspended*, tak je tato služba ignorována.

DeclareResource(⟨***jméno systémového zdroje***⟩)

DeclareCounter(⟨***jméno čítače***⟩)

DeclareAlarm(⟨***jméno alarmu***⟩)

GetResource(***ResourceType*** ⟨***ResID***⟩): tato služba změní hodnotu aktuální priority nahodnotu maximální dostupné priority pro správu zdrojů. Služba *GetResource* slouží jako vstup do kritické sekce kódu a bloku spouštěného některou z úloh nebo přerušovací rutinou, která může dostat systémový zdroj ⟨*ResID*⟩. Kritická sekce musí být vždy ukončena voláním funkce *ReleaseResource* uvnitř té samé úlohy nebo obsluhy přerušení.

ReleaseResource(***ResourceType*** ⟨***ResID***⟩): volání této služby zajišťuje opuštění kritické sekce kódu, která je přidělena systémovému zdroji, který je odkazován pomocí ⟨*ResID*⟩. Služba *ReleaseResource* se volá jako protějšek služby *GetResource*. Tato služba vrací úlohu nebo obsluhu přerušení do úrovně, v jaké byla před voláním odpovídající služby *GetResource*.

SendMessage(***SymbolicName*** ⟨***Message***⟩, ***AccessNameRef*** ⟨***Data***⟩): *Nefrontová WithCopy zpráva:* tato služba aktualizuje objekt zpráva ⟨*Message*⟩ s daty danými ⟨*Data*⟩ a žádosti přenosu objektu zprávy příjemci. Objekt zpráva je uzamknut během aktualizace a během přenosu dat nižší komunikační vrstvě a není aktualizován pokud již je uzamknut. *Nefrontová WihtoutCopy zpráva:* tato služba vyžaduje přenos už aktualizovaného objektu zprávy pro příjem. Tato služba také aktualizuje stav informace v objektu zpráva.

ReceiveMessage(***SymbolicName*** ⟨***Message***⟩, ***AccessNameRef*** ⟨***Data***⟩): *Nefrontová WithCopy zpráva:* tato služba doručuje data zprávy, které jsou spojené s objektem zpráva ⟨*Message*⟩. Objekt zpráva je během příjmu dat uzamknut. Tato služba také aktualizuje stav informace v objektu zpráva. *Nefrontová WihtoutCopy zpráva:* služba s tímto parametrem vrací status pouze od aplikace, která může přistupovat přímo k objektu zpráva.

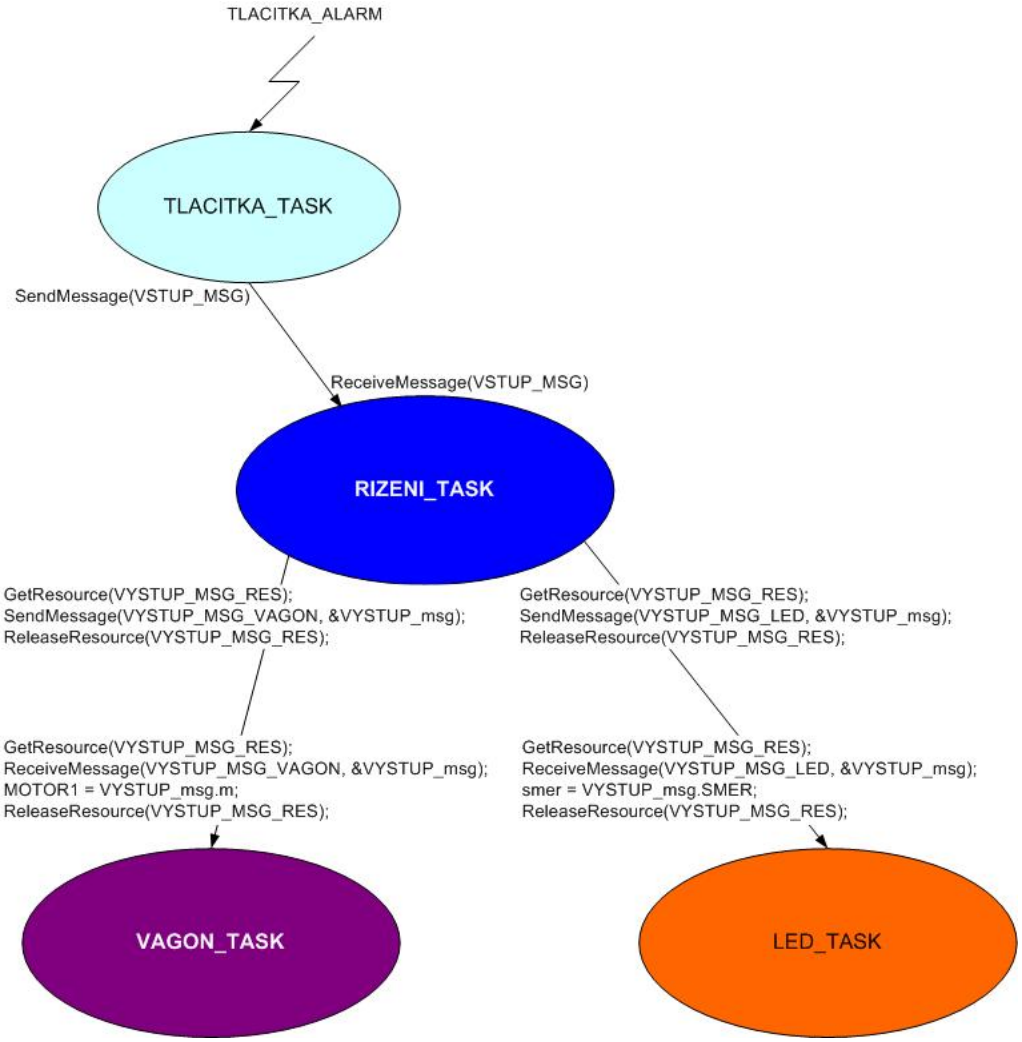
InitCounter(***CtrRefType*** ⟨***CounterID***⟩, ***TickType*** ⟨***Ticks***⟩): Nastavuje počáteční hodnotu čítače v ⟨Ticks⟩. Po tomto nastavení čítač zvýší svoji hodnotu voláním služby *CounterTrigger*. Pokud zde běží připojené alarmy, tak jejich stav zůstane nezměněn, ale čas vypršení nastane neurčitě.

6.5.2 Popis příkladu

Tato aplikace obsahuje pět úloh *INIT_TASK* s prioritou 50, *RIZENI_TASK* s prioritou 40, *VAGON_TASK* s prioritou 30, *LED_TASK* s prioritou 10 a *TLACITKA_TASK* s prioritou 0 (viz obrázek 6.5). Tento příklad provádí stejnou funkci jako příklad 6.3 s tím rozdílem, že v tomto příkladě nejsou použity žádné události a veškerá komunikace mezi úlohami je provedena pomocí zpráv, které vždy aktivují danou úlohu.

Nejprve je spuštěna úloha *INIT_TASK*, protože je ve stavu *ready* a má nejvyšší prioritu. Ta provede inicializaci systémového čítače *SYSTEMTIMER*, na který je napojen alarm *TLACITKA_ALARM*. Dále provede nastavení cyklického alarmu, který cyklicky spouští úlohu *TLACITKA_TASK*. Poté je ukončena a přechází do stavu *ready*, ve kterém již zůstává natrvalo, protože ji jiná úloha neaktivuje.

Nyní aplikace čeká na stisknutí tlačítka *run*. Jakmile je toto tlačítko stisknuto úloha *TLACITKA_TASK* posílá zprávu o stisku tlačítka *run* úloze *RIZENI_TASK*. Ta ji přijme a na základě polohy vozíku, tedy na které zarážce se naposledy nacházel určí kam se má pohybovat. Tuto informaci uloží do proměnné a pošle ji ve zprávě úloze *VAGON_TASK*. Ta ji přijme a data, kterou zpráva přenášela uloží přímo na port motoru vozíku a ten se rozjede požadovaným směrem. Úloha řízení dále ještě posílá jinou zprávu o tom, jakým směrem se vozík pohybuje úloze *LED_TASK*, která provádí signalizaci směru pohybu vozíku.



Obrázek 6.5: Popis příkladu 5.

6.6 Příklad 6: Komplexní příklad

V tomto příkladě bude ukázána komunikace mezi úlohami pomocí zpráv, použití událostí, alarmů a systémových zdrojů.

6.6.1 Použité služby operačního systému

Následují text obsahuje popis služeb operačního systému, které byly použity v tomto příkladě.

DeclareTask(⟨***jméno úlohy***⟩): *DeclareTask* slouží jako vnější deklarace úlohy, touto deklarací se sděluje deklarování funkce úlohy *TASK(⟨jméno úlohy⟩)*.

DeclareEvent(⟨***jméno události***⟩)

DeclareResource(⟨***jméno systémového zdroje***⟩)

DeclareCounter(⟨***jméno čítače***⟩)

DeclareAlarm(⟨***jméno alarmu***⟩)

ActivateTask(***TaskType*** ⟨***TaskID***⟩): tato služba zajišťuje přechod úlohy ⟨*TaskID*⟩ ze stavu *suspended* do stavu *ready*. Pokud úloha není ve stavu *suspended*, tak je tato služba ignorována.

SetRelAlarm(***AlarmType*** ⟨***AlarmID***⟩, ***TickType*** ⟨***Increment***⟩, ***TickType*** ⟨***Cycle***⟩): systémová služba okupuje alarm ⟨*AlarmID*⟩. Po ⟨*inkrementaci*⟩ čítače dojde k jeho vypršení, tím dojde k aktivaci úlohy nebo nastavení události (pouze u rozšířené události), která je k tomuto alarmu ⟨*AlarmID*⟩ přiřazena. Pokud *Cycle* se nerovná 0, tak je alarm nastaven ihned po vypršení relativní hodnoty *Cycle*. Jinak je alarm spouštěn pouze jednou. Jestliže je relativní hodnota *Increment* rovna 0, tak alarm vyprší ihned a úloha, ke které je nastaven přejde do stavu *ready* předtím, něž to systémová služba vrátí volané úloze nebo ISR.

GetResource(***ResourceType*** ⟨***ResID***⟩): tato služba změní hodnotu aktuální priority na hodnotu maximální dostupné priority pro správu zdrojů. Služba *GetResource* slouží jako vstup do kritické sekce kódu a bloku spouštěného některou z úloh nebo přerušovací rutinou, která může dostat systémový zdroj ⟨*ResID*⟩. Kritická sekce musí být vždy ukončena voláním funkce *ReleaseResource* uvnitř té samé úlohy nebo obsluhy přerušení.

ReleaseResource(***ResourceType*** ⟨***ResID***⟩): volání této služby zajišťuje opuštění kritické sekce kódu, která je přidělena systémovému zdroji, který je odkazován pomocí ⟨*ResID*⟩. Služba *ReleaseResource* se volá jako protějšek služby *GetResource*. Tato služba vrací úlohu nebo obsluhu přerušení do úrovně, v jaké byla před voláním odpovídající služby *GetResource*.

SendMessage(***SymbolicName*** ⟨***Message***⟩, ***AccessNameRef*** ⟨***Data***⟩): *Nefrontová WithCopy zpráva:* tato služba aktualizuje objekt zpráva ⟨*Message*⟩ s daty danými ⟨*Data*⟩ a žádosti přenosu objektu zprávy příjemci. Objekt zpráva je uzamknut během aktualizace a během přenosu dat nižší komunikační vrstvě a není aktualizován pokud již je uzamknut. *Nefrontová WihtoutCopy zpráva:* tato služba vyžaduje přenos už aktualizovaného objektu zprávy pro příjem. Tato služba také aktualizuje stav informace v objektu zpráva.

ReceiveMessage(***SymbolicName*** ⟨***Message***⟩, ***AccessNameRef*** ⟨***Data***⟩): *Nefrontová WithCopy zpráva:* tato služba doručuje data zprávy, které jsou spojené s objektem zpráva ⟨*Message*⟩. Objekt zpráva je během příjmu dat uzamknut. Tato služba také aktualizuje stav informace v objektu zpráva. *Nefrontová WihtoutCopy zpráva:* služba s tímto parametrem vrací status pouze od aplikace, která může přistupovat přímo k objektu zpráva.

InitCounter(***CtrRefType*** ⟨***CounterID***⟩, ***TickType*** ⟨***Ticks***⟩): Nastavuje počáteční hodnotu čítače v ⟨Ticks⟩. Po tomto nastavení čítač zvýší svoji hodnotu voláním služby *CounterTrigger*. Pokud zde běží připojené alarmy, tak jejich stav zůstane nezměněn, ale čas vypršení nastane neurčitě.

SetEvent(***TaskType***⟨***TaskID***⟩, ***EventMaskType***⟨***Mask***⟩): tato služba slouží k nastavení jedné nebo několika událostí požadované úlohy podle masky události. Pokud úloha čeká alespoň na jednu ze specifikovaných událostí, pak je uvedena do stavu *ready*. Události nespecifikované maskou zůstanou nezměněny. Na množinu událostí může být odkazována pouze rozšířená úloha, která není ve stavu *suspended*.

ClearEvent(***EventMaskType***⟨***Mask***⟩): Úloha, která volá tuto službu definuje, která událost bude smazána.

GetEvent(***TaskType***⟨***TaskID***⟩, ***EventMaskType***⟨***Event***⟩): Maska události, která je uvedena ve volání, je vyplněna podle stavu události požadované úlohy. Je vrácen aktuální stav události, ale ne maska události, na kterou úloha čeká.

WaitEvent(***EventMaskType***⟨***Mask***⟩): úloha, která volá tuto službu je uvedena do stavu *waiting* dokud není nastavena alespoň jedna událost specifikovaná maskou. Úloha zůstává ve stavu *running* pokud dojde k nastavení události v okamžiku volání služby.

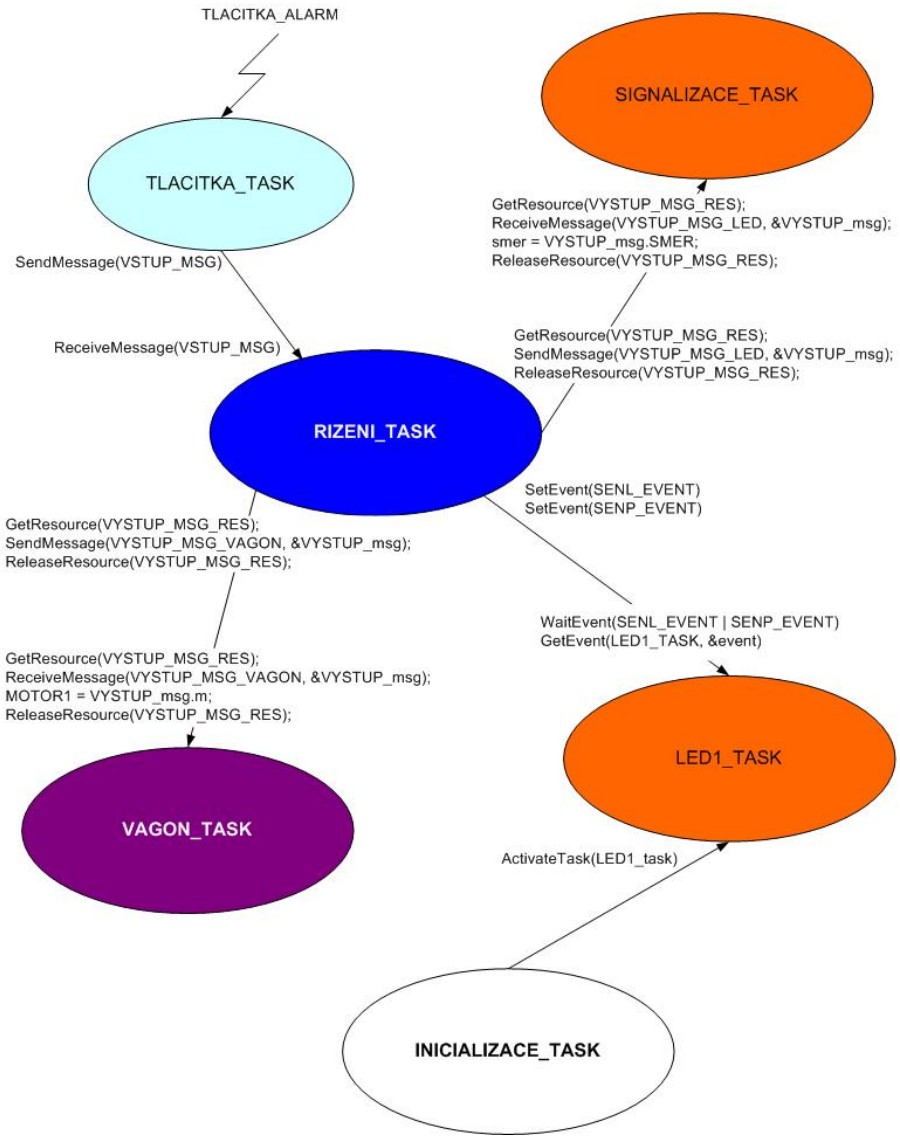
6.6.2 Popis příkladu

Tato aplikace obsahuje šest úloh *INICIALIZACE_TASK* s prioritou 50, *TLACITKA_TASK* s prioritou 0, *RIZENI_TASK* s prioritou 40, *VAGON_TASK* s prioritou 20, *SIGNALIZACE_TASK* s prioritou 10 a *LED1_TASK* s prioritou 8 (viz obrázek 6.6). Úloha *LED1_TASK* vlastní dvě události: *SENL_EVENT* a *SENP_EVENT*. Komunikace mezi ostatními úlohami je provedena pomocí zpráv

Jako první se spouští úloha *INICIALIZACE_TASK*, která je ve stavu *ready* má nejvyšší prioritu a provede nastavení systémového čítače, na který je napojen alarm *TLACITKA_ALARM*. Dále provede inicializaci úlohy *LED1_TASK*, čímž ji uvede do stavu *ready* a nastaví cyklický alarm pro cyklické spuštění úlohy *TLACITKA_TASK*. Poté je úloha

INICIALIZACE_TASK ukončena a přechází do stavu *suspended*, ve kterém již zůstává, protože ji žádná jiná úloha neaktivuje.

Nyní aplikace čeká na stisk tlačítka *run*. Po jeho stisknutí úloha *TLACITKA_TASK* pošle zprávu úloze *RIZENI_TASK*. Ta ji přijme a zjistí, že bylo stisknuto tlačítko *run*. Z polohy vozíku, tedy z hodnoty senzoru zjistí, kde se vozík naposledy nacházel a jakým má jet tedy směrem. Tuto hodnotu uloží do proměnné, kterou poté tato úloha pošle zprávou úloze *VAGON_TASK*. Ta ji přijme a data, které zpráva nesla pošle přímo na port motoru vozíku, tím se vozík začne pohybovat daným směrem. Úloha *RIZENI_TASK* rovněž pošle zprávu úloze *SIGNALIZACE_TASK* o tom, že vozík narazil na jednu ze zarážek. Ta ji přijme a zobrazí směr pohybu na vozíku. Dále se ještě v úloze *RIZENI_TASK* nastavuje událost vždy, když dojde ke změně směru vozíku, tedy když vozík narazí na jednu ze zarážek. Na tyto události čeká úloha *LED_TASK*, která po vyhodnocení, o kterou událost se jedná rozsvítí příslušný panel s diodami.



Obrázek 6.6: Popis příkladu 6.

Kapitola 7

Závěr

V rámci této diplomové práce jsem vytvořil několik vzorových aplikací v operačním systému OSEKturbo pro procesor Motorola řady HC08. Použitý hardware dodala společnost UniControls a.s. Tato společnost také dodala podklady pro vytvoření jednoduchého komunikačního modulu mezi procesorem a osobním počítačem. Tento modul jsem poté musel nejprve vyrobit a oživit, to se také po menších problémech podařilo. Největší problém byl však s použitým softwarem pro vypalování výstupního souboru do samotného procesoru, který rovněž dodala společnost UniControls. Tento software však nebyl dostatečně otestován, a proto nastal velký problém, jak program do procesoru vypálit.

Nejprve jsem vytvořil jednoduchý program bez použití operačního systému OSEK, na kterém jsem chtěl vyzkoušet naprogramování procesoru a spuštění tohoto programu. Ve vývojovém prostředí CodeWarrior jsem tedy program napsal, v simulátoru procesoru vyzkoušel a nakonec vytvořil výstupní soubor ve formátu *s19* podporovaný společností Motorola. Přistoupil jsem k vypalování programu do procesoru a v tomto okamžiku nastal problém s načítáním výstupního souboru v tomto formátu *s19*. Zjistilo se, že vypalovací software tento formát neumí dobře načíst, ale bylo mi řečeno, že formát výstupního souboru *HEX* umí načíst bez problémů. Po bližším prozkoumání vývojového prostředí CodeWarrior jsem objevil způsob, jak vytvořit výstupní soubor ve formátu *HEX*. Takovýto výstupní soubor, se mi do procesoru nakonec podařilo vypálit a program v něm spustit. Proto jsem přistoupil k vytváření aplikací v operačním systému OSEK. Opět jsem vytvořil poměrně jednoduchou aplikaci, kterou jsem chtěl vyzkoušet přímo na procesoru, ale opět nastal problém s vypálením programu do procesoru a proto jsem se rozhodl, po dohodě s vedoucím diplomové práce, že ostatní OSEK aplikace odsimuluji pouze v simulátoru procesoru ve vývojovém prostředí CodeWarrior.

OSEK aplikace, které jsem vytvořil, ukazují postup použití jednotlivých služeb tohoto operačního systému. V příloze na CD-ROM jsou tedy aplikace využívající služeb operačního systému pro správu úloh, použití služeb pro nastavení událostí, alarmů, využití čítačů, komunikace mezi úlohami pomocí zpráv a správy systémových zdrojů. V současnoti by se moje poznatky daly využít v jednom z předmětů vyučovaném katedrou a vytvořit tak například pracoviště pro výuku ve cvičení, kde by se studenti mohli prakticky s tímto operačním systémem seznámit a tím jej více rozšířit.

Pro další práci na tomto zajímavém projektu bych doporučoval buď zakoupit nebo vyrobit takový hardware, na kterém by se daly vytvořené OSEK aplikace testovat, tzn. mít k dispozici alespoň dvě desky v našem případě s procesorem Motorola řady HC08 nebo HC12 (pro tyto procesory máme zakoupený operační systém OSEK), které by mezi sebou byly spojeny sběrnicí CAN, tak jak je to aplikováno v automobilech. Nejlepší by byl

hardware přímo od společnosti, která nabízí tento operační systém a u kterého je možné provádět real-time ladění aplikace přímo na procesoru.

V této práci jsem se přesvědčil, jak těžké je seznámit se s takovýmto projektem úplně od začátku, protože jsem s takovýmto systémem neměl doposud žádné zkušenosti. Nejprve pročíst velké množství materiálu, seznámit se s novým vývojovým prostředím a poté vytvořit funkční program. Díky tomu jsem však získal obrovské zkušenosti při řešení takových to problémů.

Literatura

[1] Janáček, J.: *Distribuované systémy*, Ediční středisko ČVUT, Praha 2001.

[2] Hanzálek, Z.: *Distribuované a řídící systémy - přednášky*, ČVUT Praha 2000.

[3] Norma ISO 11898.

[4] http://www.can-cia.de

[5] Spurný, F.: *Průmuslové sítě - CAN*, Automatizaze 41/7, Praha 1998.

[6] http://www.osek-vdx.org - *Specifikace: OS22.pdf, OSEK COM V3_0.pdf, osek_nm251.pdf, OIL23.pdf*

[7] Dokumentace k OSEKturbo - *man.pdf*

[8] *OSEK/VDX : Open systems in automotive networks. Tagung Bad Homburg, 2. und 3. Februar 2000*, Düsseldorf : VDI Verlag, 2000.

Příloha: CD-ROM

Obsah CD-ROM

CD-ROM obsahuje následující data v těchto adresařích:

- Diplomka: V tomto adresáři se nacházejí zdrojové texty této diplomové práce, použité obrázky a výsledný soubor Diplomka.pdf, který vznikl překladem pomocí typografického systému PDFLaTeX.
- Vzorové příklady: Tento adresář obsahuje projekty se vzorovými aplikacemi OSEK popisovanými v kapitole 6.
- CodeWarrior - Seminar: Tento adresář obsahuje prezentaci vývojového prostředí CodeWarrior a programování procesoru Motorola řady HC08 pomocí jazyka C.
- OSEK: Zde se nachází všechen materiál co jsem nashromáždil o operačním systému OSEK.
- UniControls: Zde se nachází veškerá dokumentace k použitým deskám s procesorem Motorola HC08 a programátoru od firmy UniContols a.s.
- CAN: Tento adresář obsahuje dokumentaci k sériovému protokolu CAN.
- Help All: Tento adresář obsahuje helpy k vývojovému prostředí CodeWarrior, OSEK Builder a operačnímu systému OSEKturbo pro procesor Motorola HC08.