ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta Elektrotechnická - Katedra řídicí techniky

DIPLOMOVÁ PRÁCE

Praha 2003

Michal Krákora

Abstrakt

Tato diplomová práce se zabývá popisem a použitím real-time operačního systému OSEK a sériového komunikačního protokolu CAN. Teoretická část DP popisuje sktukturu OS OSEK a protokolu CAN. Dále uvádí praktické zkušenosti získané při práci s programovým prostředím CodeWarrior a nástrojem OSEKbuilder, které jsou součástí implementace OSEKturbo od firmy Metrowerks. Hlavním cílem praktické části bylo dokončit realizaci řídících modulů na bázi mikropočítače Motorola HC12D60 a na příkladu řídicí aplikace pracujcící v OS OSEK ověřit jejich funkčnost, tak aby mohly být použity v laboratoři distribuovaného řízení katedry řídicí techniky ČVUT FEL.

Abstract

This diploma thesis deals with an OSEK real-time operating system and serial protocol CAN. Theoretical part of this document describes OSEK operating system structure and CAN communication protocol mechanisms. Experiences with aplication development in CodeWarrior dvelopement tool and with Metrowerks OSEKbuilder are included. The goal of the practical part was to finish and validate CAN modules functionalities and abilities to be used in laboratory of Department of Control Engeneering CTU FEE.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v části „Reference“.

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).

V Praze dne ..... podpis.....

Seznam použitých zkratek

- API - „Application Programming Interface“ - služby poskytované aplikaci operačním systémem,
- RAM - random access memory - operační paměť počítače,
- ROM - read only memory - permanentní paměť,
- kbps - „kilo bit per second“ - přenosová rychlost sériového komunikačního kanálu = $1000 bit/s$,
- ČVUT - České Vysoké Učení Technické,
- CAN - controller area network - průmyslová sběrnice,
- MSCAN - Motorola scaleable Controller Area Network - hw implementace protokolu CAN od firmy Motorola,
- OS - operační systém,
- BCC - Basic Conformance Class - základní verze OS OSEK,
- FIFO - zásobníková paměť „first in first out“ přečtena je poslední uložená hodnota,
- LIFO - zásobníková paměť „last in first out“, přečtena je nejstarší uložená hodnota
- DP - diplomová práce,
- OSEK - z něměckého „Offnene Systeme und deren Schnittschstellen in die Elektronik im Kraftfarhzeug“ specifikace operačního systému,
- CD - kompaktní disk,
- DLL - „Data Link Layer“ spojovací vrstva síťového zařízení,
- DI/DO - digittální vstupy a výstupy.
- CCU - climatization control unit - řídící jednotka klimatizace.

Obsah

Seznam obrázků

1 Úvod

Mezi hlavní úkoly této diplomové práce patří seznámení s operačním systémem reálného času OSEK, seznámení s komunikačním protokolem CAN a dokončení výroby a ověření funčknosti modulů řídicích karet pracujících na bázi mikropočítače Motorola HC12D60. DP navazuje na diplomovou práci Miroslava Musila, která se zabývala vývojem a realizací výukových prostředků pro laboratoř distribuovaného řízení Katedry řídicí techniky ČVUT FEL.

Kapitola 2 se věnuje popisu sériového komunikačního protokolu CAN. Sériový komunikační protokol CAN (z angl. *Controlled Area Network*) byl vyvinut v roce 1991 firmou BOSCH. Díky vysokému zabezpečení proti chybám a především díky velmi výkonnému mechanismu detekce chyb, je vhodným nástrojem pro komunikaci v aplikacích kladoucích vysoké nároky na bezchybné doručení zpráv. Síťové uzly připojené ke sběrnici nemají adresy a nenesou informaci o síťovém okolí. To umožňuje jednoduché, takřka neomezené rozšiřování sítě.

V kapitole 3 této diplomové práce se věnuji popisu specifikace OSEK/VDX, která definuje operační systém OSEK. Operační systém OSEK je produktem, který vznikl na základě spolupráce firem působících v oblasti automobilového průmyslu sdružených v konsorcium OSEK/VDX, jako snaha vytvořit standard pro vývoj software pro elektronické řídící jednotky v automobilech. Specifikace je rozdělena do několika částí:

- OSEK Operating system - věnuje se popisu mechanismů jádra operačního systému a služeb API,
- OSEK network management - věnuje se popisu mechanismů a služeb správy sítě,
- OSEK Communication - věnuje se popisu mechanismů výměny zpráv jak v rámci jednoho procesoru, tak mezi více procesorovými jednotkami a
- OSEK System generation OIL - popisuje tzv. „OSEK implementaion language“ což je jazyk sloužící ke strukturovanému popisu nastavení jednotlivých systémových objektů, ze kterých je vybudována aplikace (úlohy, události atd.).

Zkratka OSEK vychází z německého „**O**ffnene **S**ysteme und deren Schnittstellen fur die **E**lektronik im **K**raftfahrzeug.“, což volně přeloženo znamená „Otevřené systémy a související zařízení v automobilové elektronice“ a představuje skupinu výrobců soustředěných kolem katedry IIIT německé university v Karlsruhe. Zkratka VDX (**V**ehicle **D**istributed e**X**ecutive) což volně přeloženo znamená „Distribuované řízení v automobilech“ je názvem sdružení francouzkých firem původně zcela nezávisle pracujících na tomtéž problému. V současné době jsou členy sdružení OSEK/VDX firmy jako např. Adam Opel, BMW, Daimler-Chrysler, Rober Bosch, Siemens Volkswagen atd.

Vzhledem k tomu, že výkonnými členy automobilových řídicích jednotek jsou ve valné většině 16-ti a 8-mi bitové mikroprocesory vybavené malou operační pamětí, je operační systém OSEK navržen s maximálním ohledem na optimalizaci využívaných systémových zdrojů cílových sytémů.

Specifikace operačního systému OSEK nedefinuje fyzikální vlastnosti a reprezentaci přenosového kanálu. Nejčastěji používanou komunikační vrstvou je sběrnice CAN.

Praktická část DP se zabývá vývojem distibuované řídicí jednotky automatické klimatizace za použití vývojového prostředí CodeWarrior a konfiguračního nástroje OSEKbuilder, který je součástí implementace OSEKturbo od firmy Metrowerks. Vzhledem k tomu že implementace OSEKturbo obsahuje pouze služby pro komunikaci vrámci jednoho procesoru, naprogramoval jsem knihovnu pro inicializaci a ovládání obvodu řadiče msCAN12 (viz kap. 4) osazeného na mikropočítači Motorola 68HC12D60, která umožňuje odesílat a přijímat zprávy prostřednictvím sítě CAN. Kapitola 8 obsahuje popis řídicího algoritmu a navíc obsahuje praktické

zkušenosti s vývojem aplikace a s definicí systémových objektů v prostředí CodeWarrior a OSEKbuilder.

V experimentu jsem se pokusil ovlivnit chod řídicí jenotky připojením na komfortní okruh vozidla Škoda Fabia. Výsledek experimentu je uveden v závěru diplomové práce.

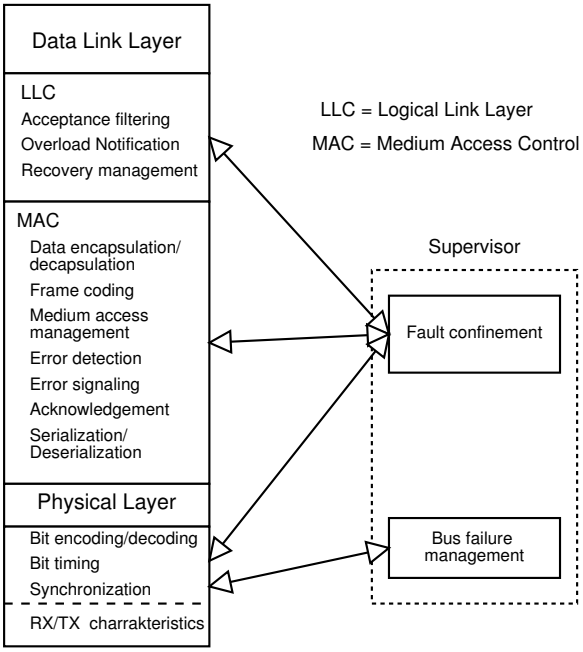
2 Komunikační protokol CAN

Tato kapitola byla zpracována na základě specifikace CAN 2.0A [8] a 2.0 B [9].

Specifikace CAN popisuje sériový komunikační protokol vyvinutý firmou BOSCH, jako prostředek určený pro komunikaci mezi elektronickými řídícími jednotkami v automobilech. Sběrnice CAN byla navržena s maximálním důrazem na zabezpečení přenášené informace proti chybám a proto splňuje i ty nejvyšší nároky kladené na přenos a doručení zpráv v časově kritických aplikacích. Vysoká míra zabezpečení a výkonný systém detekce chyb z tohoto protokolu činí ideální protředek pro komunikaci v systémech reálného času.

Vzhledem k širokému sortimentu zařízení podporujících tento komunikační protokol se z CAN v současnosti stává cenově výhodný prostředek pro komunikaci v časově kritických aplikacích. Mikroprocesorové obvody vybavené řadičem sběrnice CAN přímo na čipu a budiče sběrnice podporující různé fyzické vrstvy dnes již běžně patří do sortimentu předních světových výrobců polovodičové techniky (Motorola, Atmel atd.).

V současnosti je sběrnice CAN využívána předními výrobci osobních, ale i nákladních automobilů (Škoda-auto, Volkswagen, Opel, MAN,atd.) jako sběrnice pro řízení a sběr dat v motorové a v komfortní části automobilu. Mezi typické aplikace například patří komunikace mezi tahačem a návěsem u nákladních aut, řídící jednotky ABS, ovládání oken atp.



Obrázek 1: Sběrnice CAN z pohledu ISO/OSI referenčního modelu

Pro názornost lze zařízení splňující normu CAN z pohledu ISO/OSI referenčního modelu rozdělit na jednotlivé funkční vrstvy, jak je ukazáno na obrázku 1.

- fyzická vrstva definuje jakým způsobem budou jednotlivé signály reprezentující logické úrovně přenášeny, a proto řeší problémy spojené s časováním sběrnice, reprezentací logických úrovní a synchronizací zařízení. V rámci této vrstvy nejsou definovány fyzické

vlastnosti přenosového média, vysílačů a ani přijímačů, což přináší možnost optimalizovat a přizpůsobit sběrnici CAN dané aplikaci.

- podvrstva MAC (medium access control - řízení přistupu ke sběrnici) představuje „výkonné“ jádro sběrnice CAN. Tato vrstva přebírá zprávy od podřízené vrstvy LLC (Logical Link Control) a zároveň předává vrstvě LLC zprávy přijaté od nadřízených vrstev. MAC provádí filtrování zpráv(message filtering), arbitráž(arbitration), potvrzení příjmu zprávy(acknowledge) a detekci a signalizaci chyb (error detection and signaling). Činnost vrstvy MAC je kontrolována řídící částí, která ze nazývá *Fault confinement* což je subsytém schopný rozpoznat chyby dlouhodobého rázu od náhodně vzniklých přenosových chyb.

- LLC (logical link layer) - je vrstva spojená s filtrováním zpráv (message), overload notification a recovery managementem.

V současné době existují dvě specifikace normy CAN. Vzhledem k tomu že obě normy, tedy CAN 2.0A a CAN 2.0B, se od sebe liší pouze velikostí pole identifikátoru přenášené zprávy, budou dále popisovány současně s upozorněním na případné odlišnosti.

2.1 Datová Linková vrstva CAN (Data Link Layer)

Vzhledem k tomu, že všechna zařízení používají jeden sdílený komunikační kanál patří mezi hlavní funkce této vrstvy řešení problému přístupu aktivních uzlů k přenosovému médiu tak, aby nedošlo ke kolizi a tím ke znehodnocení přenášených dat. V anglické literatuře je tento pojem nazýván „Medium Acces Control“ (dále jen MAC)[1]. V rámci specifikace CAN je problém MAC vyřešen již díky vhodně volené fyzické reprezentaci logických úrovní signálu.

Logické úrovně přenášených dat musí být podle specifikace CAN reprezentovány „recesivní“ a „dominantní“ úrovní. Reprezentace dominantní úrovně musí mít takové vlastnosti, aby v případě současného výskytu recesivní a domintní úrovně, byla hodnota dominantní.

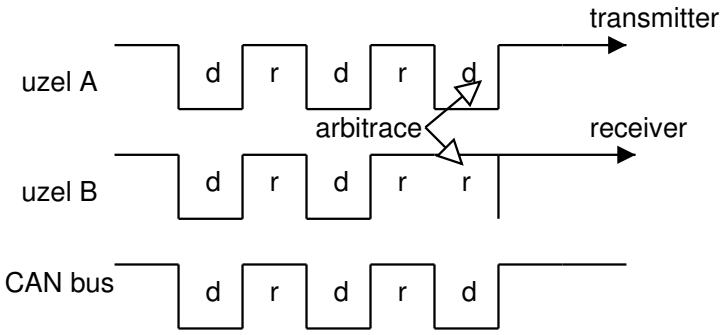
2.1.1 Arbitráž (arbitration)

Arbitráž je proces řešení přístupu k přenosovému médiu v síti CAN. Je-li sběrnice v klidovém stavu tj. neprobíhá-li na ní komunikace, může jakékoliv zařízení v síti zahájit vysílání zprávy. Pokusí-li se dvě různé jednotky vyslat data ve stejný okamžik, je tento konflikt vyřešen pomocí **arbitráže** v závislosti na poli ID přenášené zprávy a na obsahu dalších částí rámce.

Uzel „soupeřící“ o pozici vysílače v síti CAN, zároveň s vysíláním dat odposlouchává skutečný stav sběrnice. Liší-li se stav sběrnice od log. úrovně vysílané hodnoty, znamená to, že jednotka neuspěla při přístupu ke sběrnici a stává se příjemcem.

Obrázek 2 popisuje modelovou situaci, kdy dvě zařízení vysílají současně data na sběrnici. V okamžiku vyslání pátého bitu, nesouhlasí úroveň sběrnice s log. reprezentací hodnoty vysílaného bitu a proto se uzel B stává přijímačem a uzel A zůstává vysílačem. Uzel B v tomto okamžiku neuspěl v přístupu na sběrnici a pokusí se data odeslat v nejbližším možném okamžiku tj. v nejbližším okamžiku, kdy bude sběrnice vklidu.

[1] V českém překladu „Řízení přístupu k přenosovému médiu“



Obrázek 2: Přístup k přenosovému médiu.

2.2 Fyzická vrstva CAN

Jak již bylo řečeno výše, tato část specifikace popisuje přenosový kanál na úrovni fyzikální reprezentace přenášených dat. Detailní popis přesných elektrických úrovní není uveden. To přináší jistou „volnost“ při implementaci a možnost volit fyzickou vrstvu dle potřeb konkrétní aplikace.

2.2.1 Logické úrovně přenášeného signálu

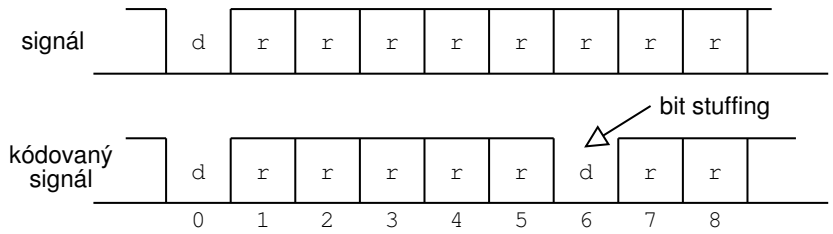
Logické úrovně přenášených dat musí být podle specifikace CAN reprezentovány „recesivní“ a „dominantní“ úrovní. Reprezentace dominantní úrovně musí mít takové vlastnosti, aby v případě současného výskytu recesivní a domintní úrovně, byla hodnota sběrnice dominantní. Pro představu, použijeme-li pro realizaci fyzické vrstvy například optické vlákno, je dominantní úrovní „světlo“ a recesivní úrovní „tma“.

2.2.2 Vysílač a přijímač

V rámci normy CAN je vysílač definován jako zařízení vysílající zprávy daného formáru do přenosového kanálu. Přijímač je z pohledu specifikace CAN zařízení schopné tyto zprávy přijímat.

2.2.3 Kódování

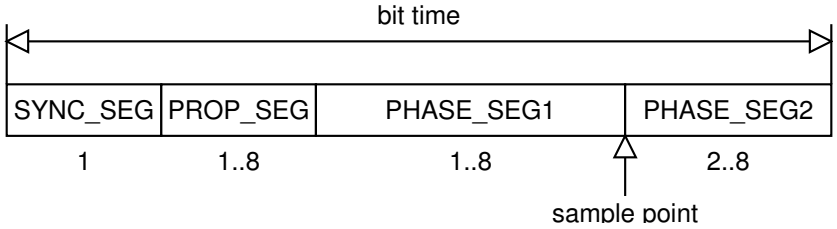
Podstatná část přenosového rámce je kódována metodou vkládání bitů tzv. **bit stuffing**. Je-li vysílačem v datech určených k odeslání zjištěna sekvence pěti po sobě následujících bitů shodné úrovně (recesivní nebo dominatní) je při vyslání za tuto sekvenci vložen bit opačné úrovně. Příklad kódování sekvence recesivních bitů je uveden na obrázku 3.



Obrázek 3: Princip kódování metodou bit-stuffing.

2.2.4 Časování sběrnice CAN

Čas pro přenesení jednoho bitu je dán převrácenou hodnotou přenosové rychlosti sběrnice. V normě CAN je časový úsek určený pro přenesení jednoho bitu rozdělen na čtyři nezávislé části (viz obrázek 4):



Obrázek 4: Struktura přenášeného bitu ve sběrnici CAN.

- SYNCHONISATION SEGMENT (`SYNC_SEG`) - synchronizační část. Hrana signálu uvnitř tohoto časového intervalu slouží k synchronizaci zařízení účastnících se komunikace.
- PROPAGATION TIME SEGMENT (`PROP_SEG`) - délka této části je úměrná době šíření signálu sběrnicí a časovému zpoždění přijímacích a vysílacích obvodů zařízení.
- PHASE BUFFER SEG1 (`PHASE_SEG1`),
- PHASE BUFFER SEG2 (`PHASE_SEG2`) tyto dvě části určují bod vzorkování tzv. SAMPLE POINT, tj. okamžik kdy bude vzorkována analogová hodnota sběrnice.

Jednotlivé části jsou složeny z definovaného počtu **časových kvant** TIME QUANTUM. Časové kvantum je atomická hodnota odvozená z frekvence oscilátoru jako celočíselný násobek „tiků“ oscilátoru.

Specifikace uvádí následující hodnoty počtu časových kvant pro jednotlivé části bitu:

- `SYNC_SEG` - 1 časové kvantum,
- `PROP_SEG` - programovatelný na $1 \ldots 8$ časových kvant,
- `PHASE_SEG1` - programovatelný na $1 \ldots 8$ časových kvant,
- `PHASE_SEG2` - programovatelný $0 \ldots 2$ časová kvanta.

Celkový čas pro přenos jednoho bitu by měl být podle specifikace programovatelný v rozsahu $8 \ldots 25$ časových kvant.

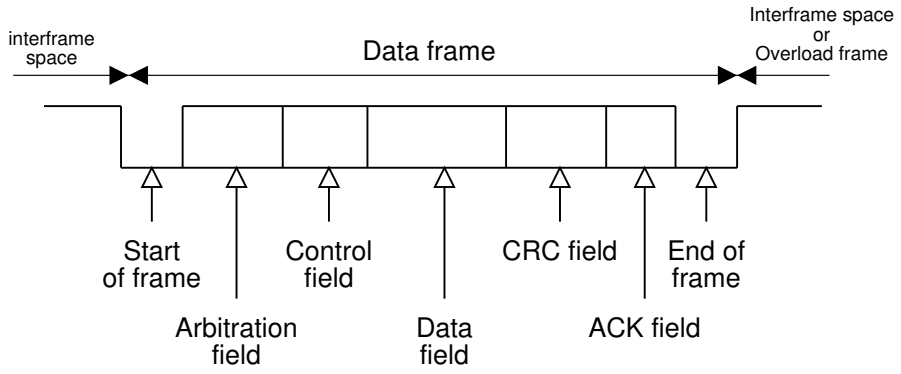
Tento způsob nastavení parametrů umožňuje velmi flexibilní změnu přenosové rychlosti sběrnice a její vzorkování s ohledem na fyzikální vlastnosti přenosového kanálu. Díky možnosti nastavení pozice bodu vzorkování, lze při návrhu zohlednit fyzikální vlastnosti sítě jako např. zpoždění vstupních a výstupních obvodů nebo fyzické dimenze sítě a tím sběrnicový systém CAN optimalizovat pro danou realizaci.

2.3 Přenosové rámce

Jak již bylo řečeno, komunikace po sběrnici CAN probíhá prostřednictvím zpráv přesně definovaného formátu tzv. **rámců**, které se dle účelu použití liší vnitřní strukturou.

2.3.1 Datový rámec (Data frame)

Datový rámec je určen k přenosu až 8-mi bajtů dat. Je rozdělen na sedm částí START OF FRAME, ARBITRATION, CONTROL FIELD, DATA FIELD, CRC FIELD, ACK FIELD a END OF FRAME (viz obrázek 5).



Obrázek 5: Datový rámec CAN.

START OF FRAME
Datový rámec začíná polem START OF FRAME, které se skládá ze dvou bitů recesivní a dominantní úrovně. Hrana dominantního bitu slouží k synchronizaci přijímacích zařízení k vysílači.

ARBITRATION
Pole arbitration je rozděleno do dvou částí:

- **ID** - složeného z 11 (CAN 2.0A) nebo 29 bitů (CAN 2.0B) představujících unikátní identifikátor přenášené zprávy v rámci celé sítě s podmínkou, že sedm nejvyšších bitů ID10 - ID4 nesmí nabývat recesivní úrovně
- **RTR** bitu, který slouží k idetifikaci datového a vzdáleného rámce. Má-li RTR bit „recesivní“ úroveň znamená to, že přenášený rámec je datový. Není-li tomu tak, a jeho hodnota je dominantní, identifikuje tzv. REMOTE rámec, jehož stavba je popsána v následujícím odstavci.

CONTROL FIELD
Další částí přenosového rámce je tzv. **control field** skládající se ze šesti bitů z nichž čtyři bity **data length code** obsahují délku pole přenášených dat a dva bity jsou nevyužity. Pole DATA LENGHT CODE může nabývat hodnot 0..7 dle počtu datových bytů v datovém poli rámce. Hodnoty DLC a odpovídající počet přenášených bytů uvádí tabulka 1.

DATA FIELD
Toto pole obsahuje 8 bajtů, které tvoří prostor pro přenášená data. Počet přenášených bytů je uveden v poli DATA LENGTH FIELD:

CRC FIELD
Toto pole je roděleno na dvě části:

- **CRC SEQUENCE** () - pole obsahuje zbytek po dělení všech předcházejících částí polynomem

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + 1$$

Počet bajtů	DLC3	DLC2	DLC1	DLC0
0	d	d	d	d
1	d	d	d	r
2	d	d	r	d
3	d	d	r	r
4	d	r	d	d
5	d	r	d	r
6	d	r	r	d
7	d	r	r	r
8	r	d	d	d

Tabulka 1: Hodnoty DLC kde r - recesivní úroveň, d - dominantní úroveň.

- **CRC DELIMITER** - slouží k oddělení CRC od následující části rámce. Ve skutečnosti trvá takovou dobu, kterou potřebují přijímací zařízení pro výpočet a ověření CRC přijaté zprávy.

ACK FIELD

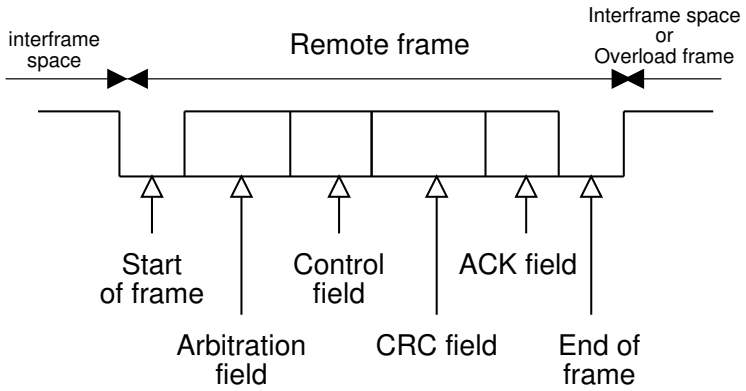
Skládá se ze dvou bitů ACK SLOTu a ACK DELIMITERu. Vysílač v datovém poli ihned po poli CR vysílá dva recesivní bity. Je-li v systému přítomna jednotka, která úspěšně přijala data, odpoví během bitu ACK SLOT tak, že změní úroveň sběrnice na dominatní hodnotu. Vysílací zařízení se tímto způsobem dozví, že byla alespoň jedné jednotce úspěšně doručena vyslaná zpráva. Nedojde-li k odpovědi, pokusí se vysílač odeslat data znovu. ACK DELIMITER musí být složen z recesivní úrovně, tak aby spolu s pole CRC DELIMITER ohraničoval prostor pro ACK SLOT.

END OF FRAME

Každý datový nebo vzdálený rámec je ukončen sedmi bity recesivní úrovně.

2.3.2 Vzdálený rámec (Remote frame)

Vyslání remote rámce znamená žádost jednotky o vyslání datového rámce se stejným identifikátorem od jakékoliv vzdálené jednotky (nódu) v síti.



Obrázek 6: Remote rámec CAN.

Vzdálený rámec se skládá ze šesti částí (viz obrázek 6) START OF FRAME, ARBITRATION FIELD, CONTROL FIELD, ACK FIELD a END OFFRAME. Složení jednotlivých částí je shodné jako u datového rámce (viz předchozí odstavec) s jediným rozdílem, a to tím, že RTR bit v ARBITRATION FIELD má dominantní úroveň.

Jak bude později vysvětleno, je tím určena priorita vzdáleného rámce před datovým. Existují-li dvě jednotky, které se pokusí současně vyslat v jednom případě datový a v druhém vzdálený rámec se stejným identifikátorem v poli ID FIELD, uspěje v arbitračním mechanismu jednotka vysílající remote rámec.

2.3.3 Chybový rámec (Error frame)

Chybový rámec (angl. Error frame)je složen ze dvou částí. První část vznikne jako superpozice tzv. ERROR FLAGů, které jsou vyslány v případě kdy jakékoliv zařízení detekuje chybu a pole ERROR DELIMITER, což je oddělovač chybového rámce.

ERROR FLAG

Zařízení účastnící se komunikace v síti se s ohledem na hodnotu vnitřních chybových čítačů může nacházet v jednom ze dvou stavů ERROR ACTIVE nebo ERROR PASSIVE. Je-li uzel ve stavu ERROR ACTIVE vysílá v případě detekce chyby na sběrnici, chybový rámec obsahující ERROR FLAG tvořený šesti bity dominantní úrovně a je-li ve stavu ERROR PASSIVE vyšle šest bitů recesivní úrovně.

Vysláním ACTIVE ERROR FLAGU aktivním uzlem se ostatní uzly v systému dozví, že nastala chyba a začnou také vysílat chybové rámce. Na lince tedy superpozicí vznikne sekvence dominantních úrovní. Její délka může být od šesti do dvanácti bitů.

Je-li stanice v pasivním stavu a detekuje-li chybu v přenosovém kanálu snaží se to sdělit ostatním stanicím tím, že vyšle PASSIVE ERROR FLAG a čeká na šest bitů shodné úrovně. Dojde-li k jejich detekci je PASSIVE ERROR FLAG dokončen.
ERROR FLAG porušuje tzv. „bit stuffing“ (viz kódování), který je aplikován na části START FIELD až CRC DELIMITER datového a vzdáleného rámce, popřípadě porušuje konzistenci polí ACK FIELD nebo END OF FRAME.

ERROR DELIMITER

Toto pole se skládá z osmi bitů recesivní úrovně. Po vyslání ERROR FLAGU stanice vyšle recesivní bity a sleduje sběrnici dokud nedojde k zachycení recesivní úrovně a tím k ukončení chybového rámce (superpozicí může dojít k tomu, že na sběrnici bude stále dominantní úroveň). Nedojde-li k detekci recesivní úrovně, je znovu vysláno osm recesivních bitů. Celý postup se opakuje dokud nenastanou podmínky pro ukončení chybového rámce.

2.3.4 Přetěžovací rámec (Overload frame)

Přetěžovací rámec se skládá z části OVERLOAD FLAG a OVERLOAD DELIMITER. Jednotka vyšle přetěžovací rámec v případě, že:

- to vyžaduje vnitřní stav přijímače aby bylo zpožděno vyslání dalšího vzdáleného nebo datového rámce nebo,
- dojde-li k detekci dominatního bitu v průběhu pole INTERMISSION.

Vyslání přetěžovacího rámce z důvodu plynoucího z první podmínky je možné pouze v okamžiku, kdy má být vyslán první bit pole INTERMISSION, zatím co k vyslání na základě druhé podmínky může dojít bezprostředně po detekci bitu dominantní úrovně.
Ke zpoždění vyslání dat nebo vzdáleného požadavku mohou být použity až dva přetěžovací rámce.

OVERLOAD FLAG
Je složen ze šesti bitů dominantní úrovně. Díky svému složení porušuje kódování bit-stuffing a tím konzistenci pole ITERMISSION. Detekují-li ostaní stanice účastnící se komunikace přetěžovací rámec odpoví vysláním pole OVERLOAD FLAG.

OVERLOAD DELIMITER
Skládá se ze šesti bitů recesivní úrovně. Jeho zpracování je vemi podobné zpracování pole ERROR DELIMITER. Bezprostředně po vyslání přetěžovacího rámce stanice sledují stav sběrnice, dokud nenastane přechod z dominatní do recesivní úrovně. V tomto okamžiku je zřejmé, že všechny stanice dokončily vyslání pole OVERLOAD FRAME a vyšlou navíc sedm recesivních bitů.

2.3.5 Mezirámcový oddělovač (Interframe spacing)

Narozdíl od přetěžovacích a chybových rámců předchází vyslání datových a vzdálených rámců vyslání tzv. mezirámcového oddělovače, který obsahuje pole INTERMISSION, BUS IDLE a pro chybově neaktivní (error passive) stanice, které byly v předchozím kroku vysílači navíc pole SUSPEND TRANSMISSION.

INTERMISSION
V průběhu tohoto pole nemá žádná stanice oprávnění začít přenos datového nebo vzdáleného rámce. Jedinou povolenou operací je signalizace přetížení (vyslání přetěžovacího rámce).

BUSS IDLE
Délka tohoto pole není přesně specifikována. V jeho průběhu je sběrnice připravena pro přenos jakéhokoliv z komunikačních rámců. Nalézá-li se v systému zařízení, které neuspělo v přístupu na sběrnici v předchozím kroku, začíná přenos bezprostředně po ukončení pole INTERMISSION. Detekce dominantní úrovně na sběrnici je považována za začátek přenášeného rámce.

SUSPEND TRANSMISSION
Po té co jednotka v pasivním chybovém stavu (error passive) odeslala zprávu a před tím než začne zjišťovat zdal je sběrnice volná pro přenos dalších dat, odešle osm bitů recesivní úrovně. Detekuje-li stanice během této doby přístup cizí jednotky ke sběrnici (vyslání nové zprávy), stáva se automaticky příjemcem.

Poznámka: Na části START OF FRAME, ARBITRATION FIELD, CONTROL FIELD, DATA FIELD a CRC SEQUENCE datového a remote rámce je použito kódování bit-stuffing popsané v odstavci 2.2.3. Zbývající části rámců CRC DELIMITER, ACK FIELD a END OF FRAME stejně tak jako těla chybových a přetěžovacích rámců nejsou touto metodou kódování ovlivněny.

2.4 Systém zabezpečení proti chybám

2.4.1 Detekce chyb

Specifikace CAN popisuje pět druhů chyb, které mohou při přenosu rámce nastat:

- **BIT ERROR - chyba bitu** - jednotka vysílající na sběrnici tj. TRANSMITTER - vysílač, současně monitoruje stav přenosového kanálu. Dojde-li k situaci kdy vysílací jednotka detekuje jinou úroveň, než právě vysílaného bitu, nastane chyba bitu. Vyjímku tvoří případ, kdy je sběrnice vysílačem buzena na recesivní úroveň a detekována je dominantní hodnota v průběhu pole ARBITRATION FIELD ovlivněného bit-stuffingem nebo v průběhu pole ACK SLOT. Vysílá-li jednotka pole PASSIVE ERROR FLAG o detekuje-li dominantní úroveň, není tento stav interpretován jako chyba bitu.
- **STUFF ERROR - chyba vkládání** -vyskytne-li se na sběrnici v průběhu vysílání bitových polí podléhajících kódování, sekvence šesti po sobě následujících bitů shodné úrovně, je tento stav interpretován jako chyba vkládání.
- **CRC ERROR - chyba kontrolního součtu** - pole CRC obsahuje výsledek výpočtu CRC pro určitou část přenášeného rámce. Přijímač provede výpočet CRC podle stejného algoritmu jako vysílač a porovnáním obsahu pole CRC s vypočtenou hodnotou ověří zda byla přijmuta nepoškozená data. V případě, že obě hodnoty CRC nesouhlasí, je to interpretováno jako chyba CRC.
- **FORM ERROR - chyba formátu** - v případě že části přenášeného pole definovaného formátu obsahují jiný než definovaný počet bitů,
- **ACKNOWLEDGEMENT ERROR - chyba nedoručení zprávy zprávy** - tato chyba nastane nedojde-li vysílačem k detekci bitu s dominantní úrovní v průběhu pole ACK SLOT. Znamená to, že vyslaná zpráva nebyla doručena ani jednomu příjemci.

2.4.2 Signalizace chyb

Dojde-li k detekci jedné z chyb uvedených v předchozím odstavci, oznámí to jednotka vysláním sekvence ERROR FLAG popsané v odstavci 2.3.3. Chybově aktivní „error active“ jednotka v případě detekce chyby vysílá tzv. ACTIVE ERROR FLAG a chybově pasivní „error passive“ jednotka vysílá PASSIVE ERROR FLAG. Je-li zachycena chyba formátu, bitu, vkládání nebo nedoručení začne vyslání ERROR FLAGU na pozici následujícího vysílaného bitu. V případě chyby kontrolního součtu, začne v případě, že nebyla detekována jiná chyba, vyslání pole ERROR FLAG ihned po odeslání pole ACK DELIMITER.

2.4.3 Chybové stavy CAN zařízení

S ohledem na systém detekce a siganlizace chyb se může jednotka účastnící komunikace v síti nacházet v jednom ze tří stavů:

- **error active** - aktivní stav,
- **error passive** - pasivní stav a
- **bus off** - odpojeno od sběrnice.

Chybově aktivní zařízení se plně podílí na komunikaci a v případě detekce chyby vysílá ERROR ACTIVE FLAG. Oproti tomu chybově pasivní zařízení v případě detekce chyby vyšle ERROR PASSIVE FLAG. Jednotka odpojená od sběrnice se nesmí žadným způsobem podílet na komunikaci a musí být odpojena od sběrnice, což znamená, že výstupní obvody budiče musí být ve stavu vysoké impedance.

Zařízení přecházejí mezi výše uvedenými stavy na základě hodnot dvou vnitřních čítačů

- **TRANSMITTER ERROR COUNTER** - čítač chyb vysílače,
- **RECEIVER ERROR COUNTER** - čítač chyb přijímače.

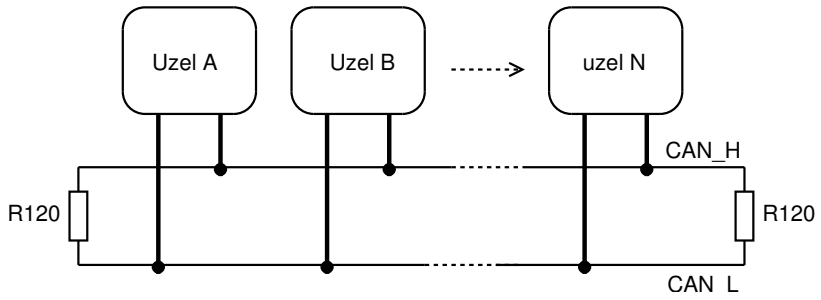
Hodnoty obou čítačů jsou modifikovány na základě chyb detekovaných při příjmu nebo vysílání zpráv. O kolik se mění stavy čítačů a za jakých podmínek k těmto změnám dochází jsou uvedeny v [8].

2.5 Používané fyzické vrstvy

2.5.1 CAN Hi-speed - popis

Fyzická vrstva CAN Hi-speed (vysokorychlostní CAN) je popsána v normě ISO 11898-2.

Přenosová linka se v minimálním případě skládá ze dvou vodičů `CAN_H` a `CAN_L` na obou koncích spojenými ukončovacími rezitory 120Ω, které představují hodnotu charakteristické impedance vedení a zamezují vzniku odrazů na vedení(viz obrázek 7).



Obrázek 7: Fyzická vrstva CAN.

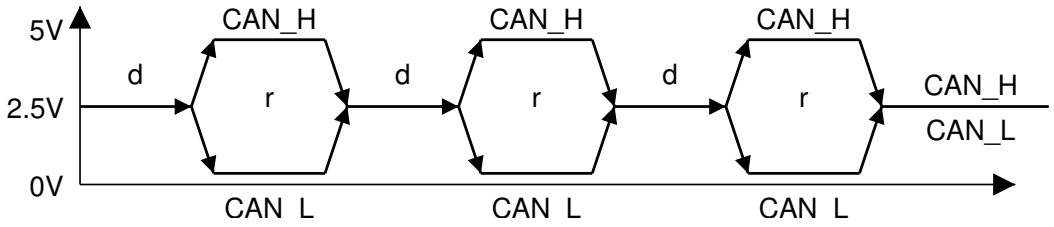
Oba vodiče `CAN_H` a `CAN_L` mají v dominantním stavu shodné napětí proti zemi(typicky 2.5V). V recesivním stavu se zvýší napětí vodiče `CAN_H` proti zemi a naopak napětí vodiče `CAN_L` se sníži(viz obrázek 8).

Typické parametry fyzické vrstvy CAN Hi-speed:

- hodnota ukončovacích rezistorů 120Ω,
- maximální délka sběrnice 40 metrů při max. přenosové rychlosti 1Mbps,
- při rychlosti 50kbps délka až 1km.

2.5.2 CAN Low speed - popis

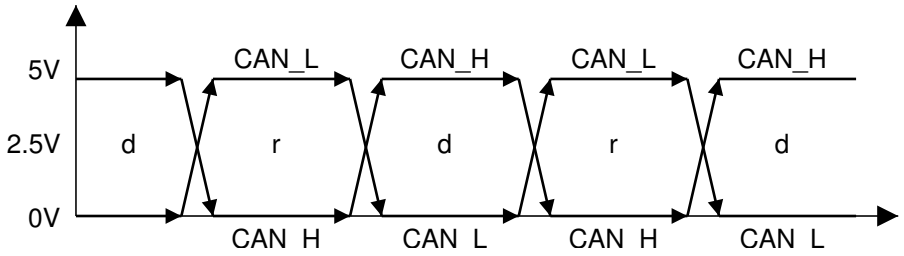
Fyzická vrstva „CAN low speed“ definuje fyzikální reprezentaci a vnitřní strukturu vysílače a příjímače. Popis fault tolerant CAN je uveden v normě ISO 11519-3.



Obrázek 8: Fyzická vrstva CAN Hi-speed.

Hlavní předností fyzické vrstvy CAN low-speed je schopnost přenosu signálu i v případě, kdy dojde ke zkratu nebo k přerušení jednoho z vodičů `CAN_H` nebo `CAN_L`. Napěťová úroveň signálu je pak měřena proti zemi baterie (kostře).

Sběrnice podle normy ISO 11529-3 může nabývat buď recesivní, nebo dominantní úrovně. Pro odlišení jednotlivých stavů sběrnice je měřeno diferenční napětí mezi vodiči `CAN_H` a `CAN_L`. V recesivním stavu je vodič `CAN_H` na nižší úrovni potenciálu než`CAN_L`, což vede na záporný rozdíl napětí V_{diff} mezi oběma vodiči. V dominantním stavu je vodič `CAN_H` na vyšší úrovni a `CAN_L` na nižší úrovni napěťového potenciálu. Napětí obou vodičů a výsledný logický stav sběrnice popisuje obrázek 9.



Obrázek 9: Fyzická vrstva CAN low speed.

Pro názornost bych rád uvedl typické parametry fyzické vrstvy CAN low-speed:

- hodnota zakončovacího rezistoru 100Ω,
- maximální počet připojených stanic v síti ≥20,
- max. délka sběrnice 40m při max. přenosové rychlosti 125kbps.
- max. délku sběrnice lze samozřejmě zvětšit užitím menší přenosové rychlosti.

2.6 Nadstavby sběrnice CAN

V současné době existuje několik definic vyšších vrstev, které umožňují jednodušší použití a vývoj systémů využívajících sběrnici CAN. Mezi nejužívanějšími bych rád uvedl:

- CAN-open standard EN50325-4,
- DeviceNet standard EN50325-2.

Bližší informace a funkční popisy jsou dostupné na webovských stránkách sdružení CiA - CAN in Automation www.can-cia.org, kde lze získat i přesné specifikace odpovídajících norem.

3 Operační systém OSEK

Specifikace OSEK/VDX popisuje systémové prostředí, které umožňuje účelné využívání systémových zdrojů v programovém vybavení řídících jednotek používaných v moderních automobilech a které je schopné splňovat vysoké nároky na vykonávání aplikačních požadavků v reálném čase - operační systém OSEK.

OS OSEK poskytuje potřebné funkční mechanismy pro vývoj a chod událostmi řízených systémů pro řízení. Specifikované služby operačního systému tvoří základ, který umožňuje integraci programových modulů vytvořených různými výrobci.

Aby bylo možno reagovat na specifické vlastnosti konrétních řídících jednotek dané jejich výkonem a snahou maximálního využití systémových zdrojů, není hlavním cílem 100% dosažení kompatibility mezi jednotlivými programovými moduly, ale jejich přímá přenositelnost. Vysoký stupeň modularity a možnosti flexibilní konfigurace dělají z operačního systému OSEK vhodný ideální systém umožňující optimální využití hardwaru, ať už je cílovým systémem jednoduchý 8-bitový mikročip nebo komplexní řídící jednotka distribuovaného řízení.

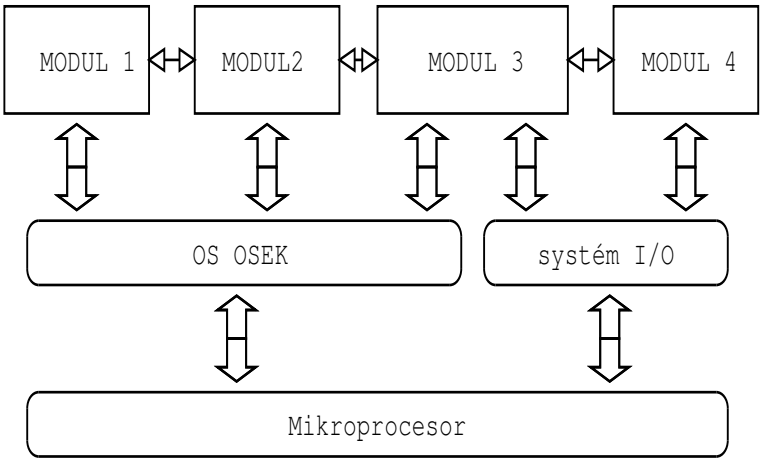
Vzhledem k nasazení operačního systému OSEK v časově kritických aplikacích, je vynecháno dynamické vytváření objektů v průběhu programu. Systémové objekty (úlohy, události atd.) jsou vytvářeny při startu systému v tzv. „fázi generování systému“. Jednotlivé komponenty a samotné jádro operačního osystému jsou vytvořeny na základě uživatelských konfiguračních instrukcí a nemohou být, až na některé výjimky, modifikovány v průběhu činnosti aplikace. Chybová hlášení, která mohou vzniknout chybným voláním systémových služeb během chodu aplikace, mohou být a často také jsou, potlačena. To má za následek zvýšení rychlosti a výkonu celé řídící jednotky.

Rozhraní mezi aplikačním softwarem a operačním systémem je zprostředkováno pomocí systémových služeb aplikačního programového rozhranní (API) a je stejné pro všechny implementace OS a různé rodiny mikroprocesorů. Pro zápis systémových služeb ve specifikaci OSEK/VDX je použit jazyk ISO/ANSI-C. Operační systém OSEK je navržen tak, aby vyžadoval minimum systémových zdrojů a je proto vhodný i pro osmibitové řady mikroprocesorů.

3.1 Přenositelnost aplikačního softwaru

Jeden z cílů projektu OSEK/VDX je docílit přenositelnosti a tím i znovuvyužitelnosti aplikačního softwaru. Proto je rozhranní mezi aplikací a OS uskutečněno pomocí přesně specifikovaného systému služeb. Použití standardizovaných systémových služeb snižuje nároky kladené na údržbu a na přenos aplikačního softwaru na jiný hardware a díky tomu snižuje náklady spojené s vývojem a odlaďováním řídících programů a jejich částí.

Aplikační program je vzhledem k logického struktuře celého zařízení umístěn nad operačním systémem (viz obr. 10) a paralelně se systémem vstupů a výstupů řídícího procesu. Vzhledem k tomu, že systém vstupů a výstupů řídící aplikace je závislý na konkrétním případě použití tj. na konkétním cílovém systému, norma OSEK jej nespecifikuje. Systém vstupů a výstupů je v logické struktuře aplikace umístěn paralelně s operačním sytémem a může být chápán jako mezivrstva mezi aplikací a hardwarem, tak jako operační systém OSEK.



Obrázek 10: Logická struktura operačního systému OSEK.

3.2 Jazyk OIL

Specifikace OSEK/VDX mimo jiné definuje jazyk pro strukturovaný popis aplikace pracující v systému OSEK tzv. Osek Implementation Langugage (OIL), který umožňuje podle přesně daných pravidel do textového souboru popsat nastavení jednotlivých systémových objektů použitých v OSEK aplikaci. Díky přesně strukturovanému a přehledému zápisu, lze pohou editací definičního souboru `*.oil` provést jednoduchou rekonfiguraci příslušného objektu a změnit tak chování objektů aplikace.

Standardní OIL soubor, lze rozdělit do dvou částí:

- definiční části - obsahuje výpis všech objektů, které je možné nadefinovat v dané implementaci operačního systému OSEK a především obsahuje jednotlivé množiny hodnot, kterých mohou parametry objektů nabývat,
- deklarační část - obsahuje seznam objektů pro danou konkrétní aplikaci s nastavením parametrů z výše uvedených množin hodnot.

3.3 Členění služeb OS

Služby operačního systému OSEK lze podle jejich funkce rozdělit do několika skupin:

- správa úloh (task management),
- synchronizace,
- správa přerušení,
- alarmy,
- systém výměny zpráv v rámci jednoho procesoru (inter processor message handling),
- systém obsluhy chyb (error handling)

3.4 Architektura OS OSEK

3.4.1 Procesní úrovně

Operační systém OSEK slouží jako koordinátor navzájem nezávislých aplikačních procesů a umožňuje jejich řízené vykonávání v reálném čase.
OSEK definuje tři procesní úrovně:

- úroveň přerušení,
- úroveň operačního systému,
- úroveň úloh.

V rámci úrovně úloh jsou procesy rozvrhovány na základě uživatelem definovaných hodnot tzv. **priorit** splňujících následující pravidla:

- přerušení má přednost před úlohou,
- procesní úroveň přerušení se skládá z jedné či více úrovní priorit přerušení,
- rutiny obsluhy přerušení mají staticky přiřazenou úroveň priority,
- přiřazení priorit rutinám obsluhy přerušení je implementačně a HW závislé,
- pro priority úloh platí, že čím větší celočíselná hodnota tím větší priorita,
- prioritní hodnoty jsou úlohám staticky přiřazeny uživatelem.

Procesní úrovně jsou definovány jako určitý rozsah celočíselných hodnot jdoucích bezprostředně za sebou dle tabulky 2. Operační systém, krom poskytování služeb aplikaci musí vhodným způsobem zajistit dodržování uvedených prioritních pravidel.

Procesní úrovně	entity
$k \ldots m$	přerušení
j	plánovač(OS)
$0 \ldots i$	úloha

Tabulka 2: Procesní úrovně a jejich řazení

3.4.2 Třídy konformity (Conformace Classes)

Různé požadavky aplikačního software na operační systém a různé vlastnosti a schopnosti konkrétních řídících jednotek vyžadují rozdílné vlastnosti operačního systému. Vzhledem k tomu, že je OS OSEK určen pro použití v řídících systémech a jednotkách, jejichž výkonnými členy jsou ve valné většině osmi a šestnáctibitové mikroprocesory disponující malou operační pamětí, je nutné optimalizovat operační systém tak, aby co nejlépe využil cílový systém a umožnil tak co nejrychlejší vykonávání řídící aplikace.

Z tohoto důvodu byly v rámci specifikace OSEK/VDX definovány tzv. **třídy konformity** (Conformation Classess - CC) operačního systému. Z pohledu uživatele, lze třídy konformity chápat jako verze, které se navzájem liší svou funkčností a tedy i rozsahem poskytovaných systémových služeb.
Jsou definovány následující třídy konformity operačního systému s rostoucí úrovní funkčnosti.

- `BCC1`(Basic Conformance Class 1[2]) - základní úlohy (viz ostavec 3.7.1 na straně 28), omezení pouze na jeden požadavek aktivace úlohy připadající na jednu úlohu, maximálně jedna úloha přiřazená jedné prioritě,
- `BCC2`(Basic Conformace Class 2) - jako BCC1, navíc dovoluje více než jednu úlohu na dané hladině priorit a povolen je i vícenásobný požadavek aktivace úlohy,
- `ECC1`(Extended Conformace Class[3]) - jako BCC1, navíc rozšířené úlohy, které mohou být i ve stavu `waiting` (viz 3.7.2 na straně 30),
- `ECC2` - jako ECC1, navíc dovoluje více jak jednu úlohu na prioritu a povolen je i vícenásobný požadavek aktivace úlohy pro základní úlohy.

Zdrojový aplikační kód napsaný pro nižší úroveň konformity je kompatibilní s vyšší úrovní. Vzájemná přenositelnost aplikačního software je možná pouze v případě, že nejsou překročeny požadavky pro danou třídu konformity. Minimální požadavky pro jednotlivé třídy konformity uvádí [2].

3.5 Mechanismus přepínání úloh - plánovač

Narozdíl od sekvenčního programování dovoluje princip multitaskingu operačnímu systému vykonávat různé úlohy „současně“. Subsystém operačního systému rozhodující o tom jaká úloha bude aktivována a bude tedy využívat procesoru se nazývá **plánovač (schleduler)**. Plánovač v operačním systému OSEK je chápán chápán jako systémový zdroj. To v praxi znamená, že může být pomocí příslušné služby OS zajištěn exkluzivní přístup k tomuto „systémovému zdroji“ a tím ovlivňován přeplánovácí proces.

Priorita úloh

Plánovač v okamžiku přerozvrhování rozhoduje o tom jaká úloha bude vykonávána na základě tzv. **priority**, což je staticky přidělená celočíselná hodnota z přesně daného intervalu čísel, kterou uživatel úloze přiřadí při vývoji aplikace. Operační systém OSEK není vybaven subsystémem správy dynamických priorit a proto nelze v průběhu vykonávávní aplikace měnit jejich hodnotu. Nejnižší úroveň z pohledu přerozvrhování má úloha s prioritou 0.

Existují-li v sytému dvě úlohy se shodnou proritou, přecházejí do stavu `running` v takovém pořadí, v jakém byly aktivovány. Vrámci jedné hladiny priorit (pouze u tříd konformity BCC2 nebo ECC2) je vytvořena FIFO fronta aktivovaných zpráv, které jsou ve stavu `waiting` a čekají na přídělení procesoru. Úloha, které byl procesor odejmut je zařazena na konec fronty zpráv příslušné hladiny priorit.

[2] Základní třída konformity
[3] Rozšířená třída konformity

Plánovač v okamžiku přerozvrhování postupuje podle následujícího algoritmu:

- vyhledá všechny úlohy ve stavu `waiting`,
- z množiny úloh ve stavu `ready` nebo `running` vybere množinu úloh s nejvyšší prioritou,
- z množiny určené v předchozím kroku najde úlohu, která čeká nejdéle a té přidělí procesor.

Ukončení úlohy

V operačním systému OSEK může být úloha ukončena pouze sama sebou. K ukončení úlohy slouží systémová služba `ChainTask` a `TerminateTask`. Použije-li úloha službu `ChainTask` sama na sebe, je nově spuštěná úloha přesunuta nakonec fronty úloh se shodnou prioritou ve stavu `ready`. Pouje-li úloha službu `TerminateTask` přejde do stavu `suspended`.

3.6 Strategie rozvrhování

Nepreemptivní rozvrhování

Strategie rozhodování je považována za nepreemptivní, děje-li se přepínání úloh na základě explicitně definovaných služeb OS v tzv. *bodech přerozvrhování*. Nepreemptivní část vykonávané úlohy s nižší prioritou odsune start úlohy s prioritou vyšší do nejbližšího bodu přerozvržení.

Body přerozvržení

Při nepreemptivním rozvrhovacím algoritmu nastane přerozvržení pouze v následujících případech:

- úspěšné ukončení úlohy službou `TerminateTask`,
- úspěšné ukončení úlohy s následnou aktivací další úlohy pomocí služby `ChainTask`,
- volání plánovače službou `Schledule`,
- přechod úlohy do stavu `waiting` službou `WaitEvent`.

Přerozvržení v explicitně daných bodech vyžaduje uložení pouze minimálního kontextu úlohy (jen několika registrů) a díky tomu klade menší paměťové nároky na operační paměť cílového systému.

Preemptivní rozvrhování

Preemptivní rozvrhování je charakteristické tím, že chod jakékoliv běžící úlohy může být, na základě vnější události, kdykoliv přerušen a provedeno nové přerozvržení. Přerušená úloha přejde do stavu `waiting` a úloha s vyšší prioritou, doposud čekající na nastalou událost, přejde do stavu `running`. Kontext ukončené úlohy je uložen a načten nový kontext právě spuštěné úlohy. Přerušení v obecném okamžiku vykonávání úlohy má za následek větší paměťové nároky, neboť je nutné krom registrů stavu procesu uložit i stav vnitřních proměnných používaných úlohou. Operační systém musí být vybaven mechanismy pro synchronizaci přístupu k systémovým zdrojům sdílených více úlohami. V případě, že je přerozvržení nežádoucí, je systém vybaven službou `GetResource()`, která umožní získání systémového plánovače jako sdíleného zdroje a tím znemožnění přerozvržení.

Body přerozvržení

- úspěšné ukončení úlohy službou `TerminateTask`,
- úspěšné ukončení úlohy s explicitní aktivací následujcí úlohy pomocí služby `ChainTask`,
- aktivace úlohy na programové úrovni úlohy pomocí služby `ActivateTask`,
- přechod úlohy do stavu waiting použitím službu `WaitEvent`,
- nastavením události na programové úrovni službou `SetEvent`,
- uvolnění používaného systémového sdíleného zdroje na programové úrovni službou `ReleaseResource`,
- návrat z úrovně přerušení na úroveň úloh.

Během vykonávání obslužné rutiny přerušení k přerozvržení nedochází. V případě nezbytnosti použití vynuceného přerozvržení systém poskytuje službu `Schedule`, která má za následek nové přerozvržení procesu.

Smíšené rozvrhování

Jsou-li v systému úlohy s preemptivním a nepreemptivním rozvrhovíáním, výsledná rozvrhovací strategie se nazývá *smíšené rozvrhnování*. V tomto případě rovrhovací strategie závisí na preempčních vlastnostech vykonávané úlohy. Je-li právě vykonávaná úloha nepreemptivní je provedeno nepreemptivní přerozvržení a podobně v případě preemptivní úlohy.

3.7 Správa úloh(Task management)

Koncept úloh

Řídící program lze ze systematického pohledu rozdělit na části, které mohou být vykonávány v závislosti na jejich real-timeových nárocích. Tyto funkční celky se nazývají **úlohy**. Úloha, což je z pohledu programátora rámec pro vykonávání služeb operačního systému a uživatelských funkcí, má přidělenou vlastní paměť tzv. **kontext úlohy**, který obsahuje kopii stavových a procesních registrů procesoru a další údaje a hodnoty nutné pro start, ukončení a v případě preemptivního rozvrhování pro přerušení a opětovné spuštění úlohy. Operační systém obsahuje **plánovač**, zajišťující synchronní i asynchronní vykonávání úloh. Plánovač operačního systému, na základě plánovací strategie, určuje v jakém pořadí budou jednotlivé úlohy přistupovat k procesoru, tj. kdy určuje okamžik kdy budou vykonávány.

Vrámci operačního systému OSEK mohou být definovány dva typy úloh:

- základní úlohy (basic tasks) a
- rozšířené úlohy (extended tasks).

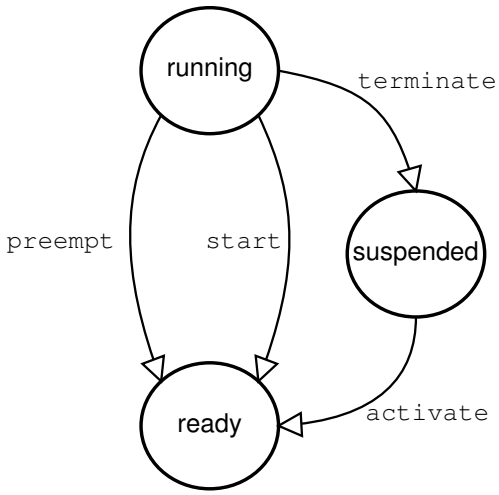
3.7.1 Základní úlohy (Basic Tasks)

Základní úloha se může nacházet pouze v jednom z následujících stavů:

- `running` - v tomto stavu se může v jednoprocesorovém systému nacházet pouze jedna úloha,

- `ready` - existují všechny funkční předpoklady pro přechod do stavu running a úloha pouze čeká na přidělení procesoru. Plánovač rozhoduje, které z úloh ve stavu ready přidělí procesor na základě priority definované uživatelem.
- `suspended` - V tomto stavu je úloha neaktivní (nemůže získat procesor). Úloha může být aktivována pomocí služby `ActivateTask()`.

Možné stavy základní úlohy a přechody mezi nimi jsou znázorněny na obrázku 11. Popis možných přechodů uvádí tabulka 3.



Obrázek 11: Stavový diagram standardní úlohy.

přechod	předchozí stav	nový stav	popis
activate	suspended	ready	úloha voláním služby OS `ActivateTask()` přejde do stavy `ready`.
start	ready	running	úloze je plánovačem přiřazen procesor a vykonávávání úlohy je zahájeno
preempt	running	ready	plánovač odebere procesor úloze a přidělí ho jiné úloze
terminate	runnign	suspended	vykonávaná úloha se sama ukončí voláním služby `TerminateTask()`

Tabulka 3: Popis přechodů mezi jednotlivými stavy základní úlohy

Základní úloha uvolní procesor pouze nastane-li jeden z následujcích případů:

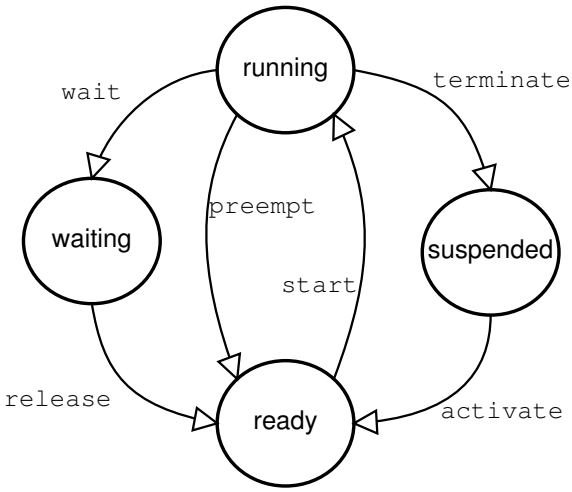
- úloha skončí,
- operační systém přepne na úlohu s vyšší prioritou,
- objeví se požadavek na obsluhu přerušení, který má za následek vyvolání obslužné rutiny přerušení.

3.7.2 Rozšířené úlohy (Extended Tasks)

Rozšířené úlohy se od základních úloh liší tím, že je u nich povoleno volat službu operačního systému `WaitEvent`, což způsobí přechod dané úlohy do stavu `waiting`. Tento stav umožňuje uvolnění procesoru a jeho přidělení úlohám s nižší prioritou bez nutnosti ukončit danou úlohu. Správa systému s rozšířenými úlohami je oproti systému se základními úlohami paměťově náročnější.

Rozšířená úloha se může nacházet v jednom z následujících stavů:

- `running` - procesor je přiřazen úloze. V tomto stavu může být pouze jedna úloha v celém systému.
- `ready` - existují všechny funkční předpoklady pro přechod úlohy do stavu `running` a úloha pouze čeká na přidělení procesoru. Plánovač rozhoduje, které z úloh ve stavu `ready` bude přidělen procesor.
- `waiting` - úloha nemůže být dále vykonávána, protože čeká na minimálně jednu událost.
- `suspend` - v tomto stavu je úloha pasivní. úloha může být aktivována voláním systémové služby `ActivateTask`.



Obrázek 12: Stavový diagram rozšířené úlohy.

Obrázek 12 zobrazuje stavové schéma rozšířené úlohy a jednotlivé přechody mezi stavy. Tabulka 4 popisuje možné přechody mezi jednotlivými stavy.

Ukončení úlohy je možné pouze tehdy, ukončí-li úloha sama sebe. Toto omezení snižuje složitost OS. Není možný přímý přechod ze stavu `suspended` do stavu `waiting`.

Aktivace úlohy

Aktivace úlohy je provedena voláním systémových služeb `ActivateTask` nebo `ChainTask`. Je-li úloha ve stavu `active`, je připravena pro vykonávání, tedy pro přechod do stavu `running`. Operační systém OSEK neumožňuje předávání parametrů úloze na začátku jejího vykonávání. Parametry mohou být úloze předány pouze pomocí systému pro výměnu dat nebo pomocí globálních proměnných.

přechod	předchozí stav	nový stav	popis
activate	suspended	ready	úloha voláním služby OS `ActivateTask()` přejde do stavy `ready`.
start	ready	running	úloze je plánovačem přiřazen procesor a vykonávávání úlohy je zahájeno
wait	running	ready	přechod do stavu `waiting` je proveden voláním systémové služby OS `WaitEvent`.
release	waiting	ready	nastala minimálně jedna událost na kterou úloha čekala
preempt	running	ready	plánovač odebere procesor úloze a přidělí ho jiné úloze
terminate	running	suspended	vykonávaná úloha se sama ukončí voláním služby `TerminateTask()`

Tabulka 4: Popis přechodů mezi jednotlivými stavy rozšířené úlohy

Násobná aktivace úlohy

V závislosti na třídě konformity může být úloha aktivována jednou a vícekrát. V případě vícenásobné aktivace je požadavek na aktivaci úlohy uložen do FIFO fronty aktivovaných úloh, které mohou přejít do stavu `running`. Maximální hodnota počtu aktivací je definována pro každou úlohu ve fázi generování systému.

Porovnání jednotlivých typů úloh

Základní úlohy nemají stav `waiting`, a proto mají synchronizační body pouze na počátku a konci úlohy. Výhodou základních úloh jsou menší paměťové nároky a menší požadavky kladené na algoritmus plánovače.

Oproti základním úlohám, nejsou-li k dispozici aktuální informace nutné pro pokračování vykonávání standardní úlohy může rozšířená úloha přejít do stavu `waiting` a počkat na událost informující o připravených datech. Rozšířená úloha je po té aktivována a pokračuje v činnosti(jsou-li k tomu splněny příslušné podmínky). Mechanismus přerozvrhování rozšířených úloh je oproti mechanismu rozvrhování standardních úloh složitější a paměťově náročnější.

3.7.3 Objekt TASK

Definice objektu TASK

Ke každé úloze v systému musí existovat odpovídající část v souboru OIL, která jednoznačně definuje parametry a chování úlohy.

```
TASK TASKSENDER {
  PRIORITY = 5;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  RESOURCE = MYRESOURCE;
  RESOURCE = SECONDRESOURCE;
  EVENT = MYEVENT;
  STACKSIZE = 64;
  ACCESSOR = SENT {
    MESSAGE = MYMESSAGE;
```

```
      WITHOUTCOPY = TRUE;
      ACCESSNAME ="MessageBuffer";
    };
  };
```

Význam jednotlivých parametrů uvádí následující tabulka.

parametr	hodnota	popis
PRIORITY	0...0x7FFFFFFF	priorita úlohy; $min = 0$
SCHEDULE	FULL,NON	definuje způsob rozvrhování úlohy
ACTIVATION	1	definuje maximální počet násobných aktivací
RESOURCE	name	jméno zdroje vlastněné úlohou
EVENT	name	jméno události vlastněné úlohou
ACCESSOR	SENT, RECEIVED	definuje způsob zpracování zprávy
MESSAGE	name	jméno zprávy odesílané nebo přijímané úlohou
WITHOUTCOPY	TRUE, FALSE	režim zpracování a uložení zprávy
ACCESNAME	string	definuje jméno proměnné přes kterou bude moci aplikace přistupovat k datům zprávy
STACKSIZE	integer	velikost zásobníku v bytech (jen u rozšířené úlohy)

Tabulka 5: Parametry objektu TASK v jazyce OIL.

Deklarace objektu TASK

K založení objektu TASK s identifikátorem `<jmeno>` a pro přiřazení funkčního těla úloze je v OS OSEK určena služba:

```
DeclareTask(<jmeno>);
```

Tabulka 3.7.3 uvádí výčet a popis systémových služeb souvisejících se systémovým objektem TASK.

služba	popis
ActivateTask	aktivuje úlohu; suspended→ready
TerminateTask	přeruší běh úlohy ready→suspended
ChainTask	přeruší právě prováděnou úlohu a spustí novou
Schedule	převede řízení na úlohu s vyšší prioritou
GetTaskId	vrátí indentifikátor běžící úlohy
GetTaskState	vrátí stav konkrétní zadané úlohy

Tabulka 6: Systémové služby objektu TASK.

Funkce tvořící tělo úlohy je ve zdrojovém kódu aplikace zadána podle následující syntaxe:

```
TASK (<jmeno>){
...
}
```

3.8 Aplikační módy (Application modes)

Aplikační módy jsou určeny pro různé režimy činnosti cílového mikroprocesorového systému. Řídící jednotka může obsahovat aplikační kód složený z několika nezávislých částí, které budou spuštěny v závislosti na režimu činnosti řídícího zařízení. Mezi tyto režimy například patří tovární test řídící jednotky, běžný provozní režim nebo ladící mód. Každá aplikace musí obsahovat alespoň jeden aplikační mód. Výběr a nastavení aplikačního módu musí být provedeno před samotným generováním systému tj. před voláním služby `StartOS` v inicializační úloze `InitTask`.

Použitím aplikačních módů lze velmi efektivně redukovat programový kód a díky tomu lépe využít omezenou pamět řídícího mikroprocesorového systému.

3.9 Zpracování přerušení (Interrupt processing)

Jedním z možných způsobů jak v programu reagovat na vnější asynchronní událost je použitím obsluhy **přerušení**. V mikroprocesoru existují dva základní druhy přerušení:

- **maskovatelné** - lze jej zakázat (periferie atp.),
- **nemaskovatelné** - nelze jej zakázat (porušení integrity paměti, výpadek napájení atp.)

Nastane-li maskovatelné přerušení a není-li zakázáno, řadič procesoru zajistí, aby v závislosti na druhu přerušení došlo v paměti programu k přesunu na místo tzv. **vektoru přerušení**, kde je uložen handler, který zajistí obsluhu přerušení. Po ukončení přerušovací rutiny se program vrátí na místo kde došlo k přerušení a pokračuje dál v činnosti.

Přerušovací vektory, což jsou vlastně adresy handlerů přerušení, obvykle bývají uloženy v chráněném místě paměti v tzv. *tabulce přerušovacích vektorů*. Pro vložení nového přerušovacího vektoru je nutné zaregistrovat nový vektror přerušení, uložením adresy přerušovacího handleru na příslušné místo v tabulce přerušení. Hondoty pozic vektorů přerušení v paměti lze získat z dokumentace ke konkrétnímu procesorovému obvodu.

Operační systém OSEK je vybaven službami pro registraci vektorů a objekty pro konstrukci obslužných rutin přerušení.

Ve specifikaci operačního systému OSEK jsou definovány tři druhy obslužbých rutin přerušení:

- ISR kategorie 1,
- ISR kategorie 2,
- ISR kategorie 3.

ISR kategorie 1
V těle funkce této obslužné rutiny přerušení **nelze** použít služeb operačního systému. Po ukončení ISR pokračuje vykonávaná úloha přesně na místě, kde došlo k přerušení. To znamená. že přerušení nijak neovlivňuje chod přerozvrhovacího procesu. ISR tohoto typu má minimální nároky na systémové zdroje (rozvrhovač a paměť RAM).

ISR kategorie 2
Operační systém OSEK umožňuje zadefinování vlastní obslužné rutiny přerušení pomocí objektu ISR. Ve fázi generování systému je dočasný přerušovací vektor nahrazen přislušným objektem ISR. V těle ISR typu 2 lze používat služeb operačního systému. Seznam služeb, které je možné použít je uveden v [2].

ISR kategorie 3
Tato obslužná rutina je svou konstrukcí shodná s ISR kategorie 1. V případě nutnosti použití služeb operačního sytému, lze použít párové služby `EnterISR` a `LeaveISR` a přejít tak dočasně na rutinu kategorie 2. Služba `LeaveISR` musí být použita jako poslední před ukončením obslužné rutiny přerušení.

Uvnitř obslužné rutiny přerušení ISR nedochází k přerozvrhování.

3.9.1 Objekt ISR

Při startu operačního systému dojde k nahrazení přerušovacích vektorů adresami oslužných rutin přerušení ISR definovaných v aplikačním kódu. Praktické zkušenosti s používáním objektu ISR jsou uvedeny v odstavci 5.3.2.

Systémové služby ISR
Tabulka 7 uvádí systémové služby související s obslužnou rutinou přerušení ISR.

služba	popis
`EnterISR`	signalizuje OS vstup do přerušovací úrovně
`LeaveISR`	signalizuje opuštění úrovně přerušení
`EnableInterrupt`	povolí přerušení
`DisableInterrupt`	zakáže přerušení
`GetInterruptDescriptor`	vrátí aktuální stav zdrojů přerušení
`DisableAllInterrupts`	zakáže všechna přerušení
`EnableAllInterrupts`	povolí všechna přerušení
`SuspendOSInterrupts`	zakáže přerušení kategorie 2 a 3
`ResumeOSInterrupts`	povolí přerušení kategorie 2 a 3

Tabulka 7: Systémové služby objektu ISR.

Vrámci aplikace je tělo obslužné rutiny ISR definováno podle následující syntaxe:

```
ISR (<jmeno>)
{
  ...
}
```

a vektor přerušení je zaregistrován pomocí systémové služby

```
DeclareISR(<jmeno>);
```

Každému objektu ISR v aplikaci musí odpovídat příslušná část v definičním OIL souboru. Následuje příklad výpisu OIL souboru:

```
ISR Handler {
  PRIORITY = 0;
  CATEGORY = 2;
  RESOURCE = ISRResource;
  ACCESSOR = RECEIVED {
    MESSAGE = messageA;
    ACCESSNAME = myBuffer;
  };
};
```

Význam parametrů a hodnoty, kterých mohou nabývat je shodný s objektem TASK.

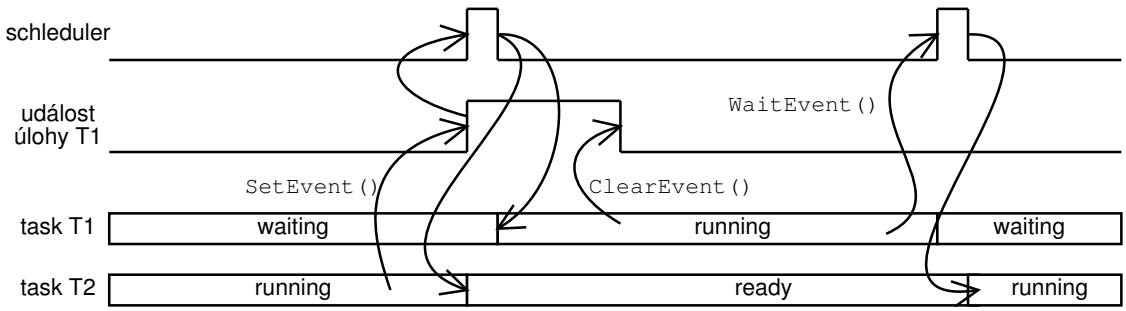
3.10 Systém událostí (Event Mechanism)

Události slouží jako synchronizační nástroje rozšířených úloh. Mimo to mohou být použity jako prostředek pro zasílání zpráv binárního charakteru (motor běží, zpráva přijmuta atp.). Události v operačním systému OSEK jsou objekty spravované operačním systémem. Nejsou to nezávislé objekty, ale jsou přiřazeny rozšířeným úlohám. Každá rozšířená úloha může být „majitelem“ určitého počtu událostí. Událost je potom jednoznačně definována majitelem a svým jménem.

Z těla jakékoliv rozšířené úlohy nebo ISR lze událost pomocí systémové služby „SetEvent“ nastavit, ale deaktivace nebo čekání až událost nastane je možná pouze z těla „majitele“ pomocí služby `ClearEvent` resp. `WaitEvent`.

Rozšířená úloha přejde ze stavu `waiting` do stavu `running`, nastala-li alespoň jedna událost, na kterou úloha čekala. Pokusí-li se rozšířená úloha čekat na událost, která již nastala a nebyla vymazána, zůstane ve stavu `running`.

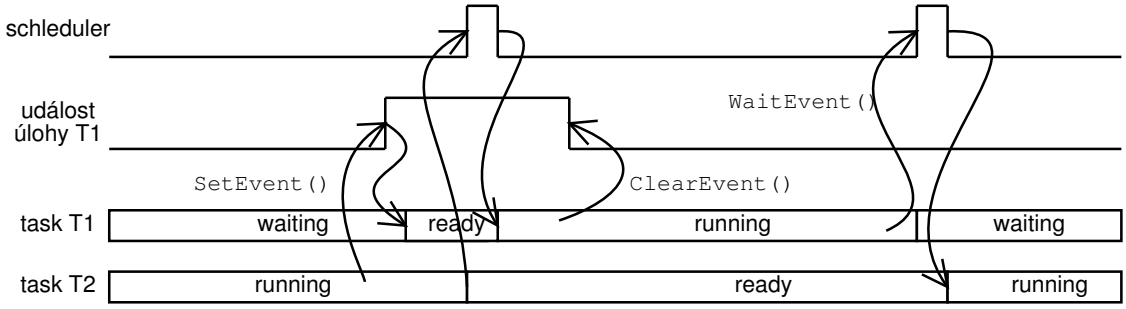
Příklad 1
Obrázek 3.10 vysvětluje synchronizaci rozšířených úloh pomocí událostí v preemptivním systému, kde úloha T1 má vyšší prioritu než úloha T2. Obrázek ilustruje pochody, které jsou ovlivněny nastavením události. Úloha T1 čeká na událost. Úloha T2 nastaví tuto událost pro T1. Plánovač se aktivuje v okamžiku nastavení události a přepne úlohu ze stavu `waiting` do stavu `ready`. Na základě vyšší prioroty úlohy T1, dojde k přerozvržení systému, tedy k přepnutí úlohy T2 do stavu `ready` a úlohy T1 do stavu `running`. Dále úloha T1 vymaže pomocí systémové služby `ClearEvent` nastalou událost. Po té úloha T1 znovu čeká na událost a úloha T2 přejde do stavu `running` a celá situace se opakuje.



Obrázek 13: Synchronizace úloh pomocí událostí v preemptivním systému.

Příklad 2

Obrázek 3.10 ukazuje synchronizaci rozšířených úloh v nepreemptivním systému, kde úloha T1 má vyšší prioritu než úloha T2. Z obrázku je patrné, že k přepnutí úloh nedojde ihned po přechodu úlohy T1 do stavy `ready`, ale až po zavolání systémové služby `Schedule` z těla úlohy T2. Vykonání úlohy T1 s vyšší prioritou je tedy pozdrženo až do nejbližšího bodu přerozvržení.



Obrázek 14: Synchronizace úloh pomocí událostí v nepreemptivním systému.

3.10.1 Objekt EVENT

Pro zaregistrování události `<jmeno>` a vytvoření objektu EVENT v aplikaci je nutné použít službu
`DeclareEvent(<jmeno>)`
a do definičního souboru OIL přidat nádleující text:

```
EVENT <jmeno> {
  MASK = 0x01;
  }
```

Jak bylo popsáno výše událost musí mít svého „majitele“, což znamená, že musí existovat v systému úloha, jejíž definice v programu OIL obsahuje v poli `EVENTS =` identifikátor příslušné události.

Operační systém OSEK disponuje následujícími systémovými službami pro využití systému událostí.

služba	popis
`SetEvent`	nastaví událost se shodnou zadanou maskou
`ClearEvent`	vymaže událost dle zadané masky
`GetEvent`	vrátí aktuální stav událostí příslušné úlohy
`WaitEvent`	přesune volající úlohu do stavu `waiting`

Tabulka 8: Systémové služby objektu EVENT.

3.11 Správa systémových zdrojů (Resource Management)

Systémové službu správy systémových zdrojů se používají k zajištění přístupu úloh ke sdíleným zdrojům, jakými například jsou sdílená globální data, periferie atd.
Mezi její hlavní funkce patří:

- zamezení a ošetření situace, kdy dvě úlohy přistupují k jednomu systémovému zdroji,
- zamezení tzv. inverze priorit, to znamená situace, kdy úloha s vyšší prioritou čeká na ukončení úlohy s prioritou nižší, neboť ta právě „okupuje“ systémový zdroj vyžadovaný čekající úlohou,
- aby přístup k sytémovým zdrojům nikdy neskončil ve stavu `waiting`.

3.11.1 OSEK protokol mezních priorit (OSEK ceeling protocol)

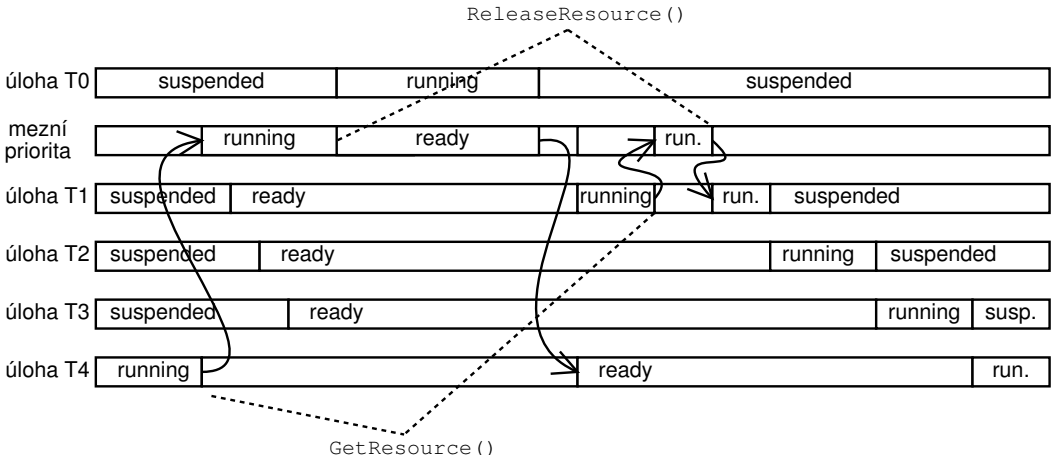
Specifikace operačního sytému OSEK definuje tzv. „Protokol mezních priorit“, který řeší situaci, při které by se úloha nebo ISR pokušela přistoupit k systémovému zdroji používanému jinou úlohou nebo ISR. V případě, že je systém správy systémových zdrojů použit pro koordinaci úloh a ISR, operační systém též zaručuje, že je ISR vykonána pouze v tom případě, že jsou všechny požadované systémové zdroje volné.

Z důvodu zabránění vzniku inverze priorit a deadlocku vyžaduje operační systém OSEK následující chování:

- při generování systému má každý systémový zdroj staticky přiřazenou svou vlastní tzv. „mezní“ prioritu. Mezní priorita je nastavena na úroveň nejvyšší priority ze všech úloh, které využívají daný systémový zdroj. Mezní priorita musí být současně nižší než úroveň nejnižší priority ze všech úloh, které nepoužívají systémový zdroj, a které mají prioritu vyšší, než úroveň nejvyšší priority všech úloh, které používají systémové zdroje.
- Získá-li úloha přístup k systémovému zdroji a její priorita je nižší než je mezní priorita daného systémového zdroje, priorita úlohy je zvýšena na úroveň mezní priority tohoto systémového zdroje.
- Když úloha uvolní systémový zdroj, její priorita je nastavena na její původní hodnotu, kterou měla před získáním systémového zdroje.

Úlohy, které si vyžádají již obsazený systémový zdroj neuspějí v jeho přidělení, neboť mají nížší nebo maximálně stejnou hodnotu priority jako úloha, která zdroj momentálně vlastní. Je-li systémový zdroj uvolněn, úlohy, které čekají na jeho uvolnění mohou přejít do stavu `running`. Při použití protokolu mezních priorit může dojít k časovým prodlevám u úloh, které mají nižší prioritu než je mezná hodnota systémového zdroje. Prodlevy jsou dány nejdleší dobou používání sdíleného zdroje úlohou s nižší prioritou.

Příklad
Příklad na obrázku 15 ukazuje mechanismus protokolu mezních priorit. Úloha T0 má nejvyšší prioritu a úloha T4 má nejnižší prioritu. Úlohy T1 a T4 chtějí získat systémový zdroj. Příklad ilustruje, že nedochází k žádné neomezené inverzi priorit. Úloha T1 s vysokou prioritou čeká kratší čas, než je maximální doba trvání používání systémového zdroje úlohou T4.



Obrázek 15: Příklad použití protokolu mezních priorit.

Protokol mezních priorit s rozšířením pro úroveň přerušení

Protokol mezních priorit je nutno rozšířit pro úroveň přerušení. Mezní priority systémových zdrojů, které jsou používány přerušeními jsou dány jako virtuální priority vyšší než nejvyšší priorita z priorit přiřazených všem přerušením. Výpočet mezní priority znamená jiný přístup pro systémový zdroj, který je používín pouze úlohou a jiný přístup pro systémový zdroj používaný úlohou i rutinou obsluhy přerušení. Zacházení se softwarovými prioritami a hardwarovými úrovněmi přerušení závisí na dané implementaci operačního systému. Pro implementaci protokolu mezních priorit s rozšířením pro úroveň přerušení je vyžadováno následující chování:

- Při generování systému je zdroji přiřazena statická hodnota mezní priority, jejíž hodnota je nastavena nejméně na úroveň nejvyšší priority ze všech úloh a ISR, které zdroj používají. Mezní priorita musí být současně nižší než úroveň nejnižší priority ze všech úloh nebo ISR, které nepoužívají systémový zdroj, a které mají současně prioritu vyšší než úroveň nejvyšší priority všech úloh nebo ISR používajících systémové zdroje.
- Získá-li úloha nebo ISR přístup k systémovému zdroji a její aktuální hodnota je nižší než mezní prirota systémového zdroje, je její hodnota priority dočasně zvýšena na úroveň mezní priority tohoto zdroje.
- Uvolní-li úloha nebo ISR systémový zdroj, je hodnota její priority vrácena na původní hodnotu, kterou disponovala před přidělením systémového zdroje.

Úlohy nebo ISR, které by mohly používat stejný systémový zdroj jako používá aktuálně vykonávaná úloha nebo ISR, nepřechází do stavu `running` v důsledku jejich nižší nebo stejné priority jako má aktuálně vykonávaná úloha, jejíž priorita byla dočasně změněna. Je-li požadovaný systémový zdroj uvolněn, mohou čekající úlohy přejít v nejbližším bodu přerozvrhování přejít do stavu `running`.

3.11.2 Objekt RESOURCE

Pro zaregistrování identifikátoru `<jmeno>` a vytvoření systémového zdroje je určena systémová služba

```
DeclareResource(<jmeno>);
```

Každému sdílenému zdroji použitému v aplikaci musí odpovídat příslušná část v definičním souboru OIL.

```
RESOURCE <name>;
```

Operační systém je pro mechanismus sdílení zdrojů vybaven službami uvedenými v tabulce 9.

služba	popis
`GetResource`	získání sdíleného zdroje
`ReleaseResource`	navrácení systémového zdroje

Tabulka 9: Systémové služby objektu RESOURCE.

3.12 Alarmy (Alarms)

Operační systém OSEK je vybaven systémovými službami pro zpracování periodických událostí. Zdroji takových událostí mohou být například časovače, které generují přerušení vždy v určitých pravidelných intervalech, nebo například údaje z inkrementálního úhlového snímače, generujícího přerušení vždy při úhlovém natočení o konstantní úhel a pod. Operační systém OSEK poskytuje dvouúrovňovou koncepci pro zpracování takových událostí. Periodické události jsou zaznamenávány implementačně specifikovanými čítači. Na základě stavu čítačů je postaven systém alarmů.

Čítače

Čítač je reprezentován svou hodnotou v „ticích“ a několika konstantami, které určují parametry čítače. Operační systém OSEK nedisponuje standardizovanými službami pro přímou obsluhu čítačů, ale zajišťuje nezbytné akce týkající se správy alarmů, je-li čítač rozšířen na alarm. Operační systém OSEK disponuje alespoň jedním čítačem přímo vázaným na hardwarový čítač systému (dáno specifikací OSEK/VDX).

Správa alarmů

Operační systém OSEK poskytuje služby pro aktivaci úloh nebo množiny událostí na základě doběhnutí alarmu. Doběhnutí alarmu znamená dosažení uživatelem nastavené hodnoty čítače asiciovaného s alarmem. Hodnota čítače může být definována relativně vzhledem k aktuální hodnotě čítače, pak se jedná o *relativní hodnotu* nebo absolutně a pak se jedná o *absolutní hodnotu* alarmu. Alarmy mohou být definovány jako tzv. *single alarmy*, což znamená že akce svázaná s doběhnutím alarmu bude vykonána pouze jednou, nebo jako *cyklické alarmy*, což analogicky znamená, že akce svázaná s alarmem bude vykonávána opakovaně s nastavenou frekvencí. Operační systém poskytuje služby pro vymazání alarmů a zjišťování jejich aktuálních hodnot.

S doběhnutím alarmu mohou být asociovány dvě akce:

- aktivace úlohy - při doběhnutí alarmu je aktivována úloha jako by došlo k použití služby `ActivateTask`,
- nastavení úlohy - při doběhnutí larmu je nastavena událost jako při použití standardní služby `SetEvent`.

3.13 Komunikace

3.13.1 Komunikační třídy konformity

Specifikace OSEK/VDX popisuje celý komplex mechanismů, označovaný jako OSEK COM, pro výměnu a přenos zpráv vrámci jednoho procesoru ale i komunikační sítě přes kterou komunikuje více řídících jednotek. OSEK COM používá tzv. asynchronní komunikační model, což v praxi znamená že komunikační funkce jsou vykonávány paralelně s aplikací. Mechanismy výměny správ vrámci komunikační sítě v sobě zahrnují různé služby, které umožňují zabezpečování komunikací proti chybám, neoprávněné manipulaci se zprávami, samočinné kontroly funkce sítě atd. Implementace takových mechanismů s sebou přináší vysoké nároky kladené na systémové zdroje jednotlivých řídících jednotek.

Mnoho aplikací vyžaduje pouze jistou omezenou část komunikačních služeb. Aby bylo možno efektivně využívat systémové zdroje řídících jednotek byly definovány čtyři úrovně komunikačních tříd tzv. „Communication Conformace Classes“. Komunikační třídy jsou seřazeny vzestupně podle rostoucí úrovně funkcionality

- CCCA - umožňuje použít pouze interprocesorovou komunikaci pomocí nefrontových zpráv a nedisponuje službami pro zjištění stavu správy.Obsahuje pouze systémové služby `SendMessage` a `ReceiveMessage`.
- CCCB - stejné jeko CCCA s rozšířením o služby umožňující zjistit stav zprávy `GetMessageStatus` a umožňuje využívat frontových zpráv.
- CCC0 - umožňujuje jak interprocesorovou, tak meziprocesorovou komunikaci, obsahuje vše co CCCA s rozšířením o režim vynuceného přenosu zprávy (Direct Transmition Mode).
- CCC1 - obsahuje plnou implementaci OSEK COM podle [1]

Zdrojový aplikační kód napsaný pro úroveň s nižší třídou konformity je přenositelný do úrovně s vyšší třídou.

Operační systém OSEK v implementaci OSEKturbo od firmy Metrowerks v licenci vlastněné Katedrou řídicí techniky ČVUT FEL v sobě zahrnuje pouze komunikační třídu CCCA umožňující komunikaci vrámci jednoho procesoru a to pouze pomocí nefrontových zpráv.

3.13.2 Zprávy - messages

Zpráva je softwarový kontejner pro pro aplikací specifikovaná data. Může mít v systému pouze jednoho odesílatele, ale jednoho a více adresátů. Operační systém definuje dva základní druhy správ:

- frontové zprávy a
- nefrontové zprávy.

Frontové

Frontové zprávy jsou po příjmu uloženy systémem do fronty FIFO v pořadí v jakém byly přijmuty. Data obsažená ve frontových zprávách mají událostní charakter např. zpráva o otevření a zavření dveří automobilu, zapnutí nebo vypnutí jednotky atd.

Nefrontové
Zprávy tohoto typu obsahují data informačního charakteru. Data z příchozí zprávy jsou ukládána do paměti na stále stejnou adresu. To znamená, že nově příchozí data přepisují původní hodnoty. Nefrontové zprávy lze chápat jako např. údaje z teplotního čidla.
Typ dat přenášené zprávy je pevně definován ve fázi generování operačního systému. Délka zprávu může být definována jako statická nebo dynamická.

- statická délka zprávy - slouží pro běžnou výměnu dat mezi úlohami v běžícími v operačním systému systému,
- dynamická délka zprávy - délka datového pole zprávy je nastavena dynamicky v průběhu vykonávání aplikace.

Zprávě je možné přiřadit další funkce jako například aktivaci úlohy, nebo nastavení události, které umožňují okamžité přijetí nebo odeslání zprávy. Ve specifikaci operačního systému nejsou definovány žádné služby umožňující čekání na příjem zprávy. Událostní systém je proto jediným možným mechanismem jak „blokovat“ úlohu dokud nepřijde příslušná zpráva.

Při běžné výměně dat vytváří operační systém každému adresátovi lokální kopii příchozích dat tzv. *WithCopyAccess*. Adresát pak může s přijmutými daty pracovat bez nutnosti řešení přístupu k datům a k jejich konzistence. Z důvodu optimalizace využití operační paměti je k dispozici tzv. *WithoutCopyAcces*, při kterém příjemci sdílejí jeden příchozí buffer. V tomto případě není zaručena konzistence dat a tudíž je na uživateli aby pomocí vhodných mechanismů jako např. mechanismu sdílení systémových zdrojů (semaforů) zajistil exkluzivní přístup k tomuto zdroji dat.
Volbu chování systému při příjmu zprávy je nutno definovat ve fázi generování systému.

Odeslat nebo přijmout zprávu v operačním systému OSEK lze pomocí služeb `SendMessage` a `ReceiveMessage`.

3.13.3 Objekt MESSAGE

Pro definic objektu MESSAGE v aplikaci je nutné přidat do definičního souboru OIL následující část

```
MESSAGE MsgA{
  TYPE = UNQUEUED;
  CDATATYPE = 'long int';
  ACTION = SETEVENT {
    TASK =task1;
    EVENT = eventC;
  };
};
```

Tímto jsme nadefinovali nefrontovou zprávu `MsgA`, obsahující data typu `long int` po jejímž příchodu bude aktivována událost `eventC` majitele `task1`.

Pro odeslání a příjem zpráv je operační systém vybaven funkcemi `SendMessage` a `ReceiveMessage`.

3.14 Rutiny odskoků (Hook-up routines)

Rutiny odskoků umožňují provádět uživatelské operace v době kdy operační systém OSEK provádí své vnitřní aktivity, ke kterým uživatel jinak nemá přístup. Uživatel má k dipozici následující rutiny odskoků:

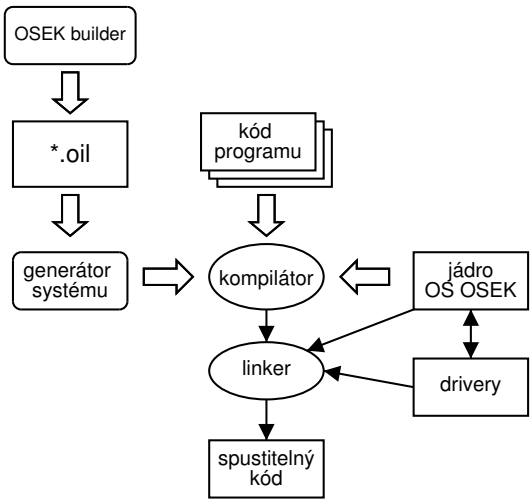
- `StarupHook` - rutina startu systému, je spuštěna bozprostředně po startu systému,
- `ShutdownHook` - rutina ukončení činnosti systému, je spuštěna ihned po ukončení běhu systému,
- `PreTaskHook` a `PostTaskHook` - rutiny ladění programů,
- `ErrorHook` - rutina chyb.

Rutiny `PreTaskHook` a `PostTaskHook` umožňují v době přerozvržení provádět uživatelem definované operace. Pro detailnější popis bych si dovolil odkázat na [2].

3.15 Generování systému

V předchozí části dokumentu jsem se velmi často odkazoval na tzv. „fázi generování systému“, což je proces při kterém uživatel vytváří uživatelský program, definuje systémové objekty, konfiguruje systémové objekty a vztahy mezi nimi a poté vše spojí se softwarovými moduly jádra operačního systému do výsledného sputitelného souboru obsahujícího strojový kód aplikace.

Fázi generování systému symbolicky popisuje obrázek 16. Uživatel vytvoří soubory obsahující zdrojové kódy a OIL soubor, který může vytvořit buď „ručně“ a nebo pomocí nějakého automatického generátoru, který je velmi často součástí konkrétní amplementace (například OSEK Builder od firmy Metrowerks). Definiční OIL soubor je zpracován generátorem systému, který podle jaho obsahu vygeneruje všechny potřebné aplikačně závislé soubory. Vygenerované soubory spolu se soubory obsahujícími uživatelské kódy a s knihovnami zkompiluje a slinkuje do spustitelné formy.



Obrázek 16: Fáze generování systému

4 Obvod msCAN12

Mikropočítač Motorola 68HC12D60 je vybaven modulem řadiče sběrnice msCAN12, který obsahuje implementaci průmyslového sériového komunikačního protokolu CAN 2.0A/B. Následující část dokumentu je věnována popisu základní struktury modulu msCAN12 a jeho nastavení. Především bych se rád zaměřil na popis a nastavení uživatelských registrů, které jsem použil v praktické části pro inicializaci a obsluhu obvodu. Pro detailní pochopení nastavení obvodu msCAN12 a úplný popis jeho vnitřních registrů bych doporučil prostudovat [6].

4.1 Základní popis

Jak již bylo řečeno obvod msCAN12 umožňuje 16-bitovému mikropočítači Motorola 68HC12 aktivní účast na komunikaci v síti CAN. Pomocí řídících registrů obvodu lze jednoduše nastavit veličiny jako například přenosovou rychlost (až na 1Mbps) nebo zamezit příjmu zpráv, jejichž identifikátory nesplňují podmínky pro průchod vstupními filtry. Stavy obvodu jako například příjem zprávy, prázdný odchozí buffer, detekce chyby v síti nebo přechod obvodu do stavu pasivního vysílače je možné detekovat pomocí maskovatelného přerušení a díky tomu nastalou situaci ošetřit pomocí příslušné rutiny přerušení.

4.2 Přijímací buffer

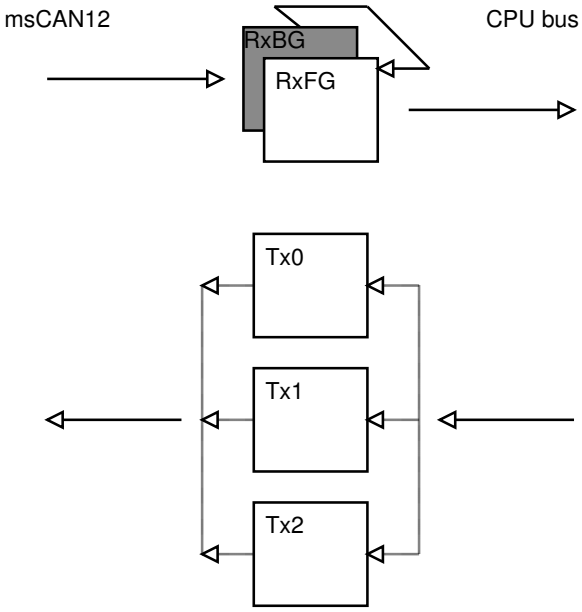
Příchozí rámec je uložen do mikropočítačem neadresovatelného třináctibytového přijímacího bufferu RxBG. Je-li příchozí rámec bezchybný (kontrola CRC a bit stuffing) a splňuje-li jeho identifikátor podmínky pro průchod přijímacími filtry obvodu msCAN12, je zkopírován do bufferu RxFG umístěného v adresovém prostoru mikropočítače (viz obrázek 17). Po uložení zprávy do bufferu RxFG nastane, není-li maskováno, přerušení od přijímacího bufferu a tak informována nadřazená aplikace o příchodu nových dat. Neprojde-li příchozí rámec vstupními uživatelsky definovatelnými filtry, nedojde k jeho zkopírování do adresovatelného bufferu a jeho obsah je přepsán nově příchozími daty.

Přijímací buffer RxFG je v paměti umístěn od adresy 0x0110 dále. Struktura prvních čtyř bytů obsahujících informaci o identifikátoru zprávy se liší podle použití standardních nebo extended identifikátorů. Při použití standardního 11 bitového identifikátoru úroveň log. 1 v bitu RTR identifikuje vzdálený rámec a bit IDE musí být vždy nastaven na hodnotu log. 0. Při použití rozšířených identifikátorů musí být bity SRR a IDE nastaveny na úroveň log. 1. Bit RTR pak plní stejnou funci jako u standardního identifikátoru, tedy slouží jako identifikátor vzdálené žádosti o data.

4.3 Vysílací buffery

Obvod msCAN12 obsahuje tři vysílací bufery Tx0, Tx1 a Tx2. Programátor tedy může v okamžiku kdy jsou odesílána data z plných zásobníků vhodným způsobem zajistit plnění prazdných bufferů. Je-li příslušní odchozí zásobník prázdný informuje o tom stavový registr odchozích zásobníků CTFLG. Stavba pole identifikátorů IDRx odpovídá struktuře přijímacího bufferu. Význam bitů RTR, SRR a IDE se stejně jako u odesílacího bufferu liší v závislosti na použitém standardním nebo rozšířeném identifikátoru a byl popsán v předchozím odstavci.

Oproti přijímacímu bufferu však na arese 0x01xD vysílací buffery obsahují pole TBPR (Transmission Buffer Priority Register) umožňující prioritní řazení bufferů při požadavku odeslání dat. Dojde-li k situaci, že všechny buffery obsahují neodeslaná data a přijde-li požadavek



Obrázek 17: Systém vstupních a výstupních zásobníků.

na jejich odeslání určí obvod msCAN12 pořadí odeslání zpráv v závislovsti na hodnotě prioritního pole TBPR. Pro další informace bych rád odkázal na[6].

4.4 Řídící registry obvodu

CMCR0

V tomto registru bych rád upozornil na bit SFTRES, který je-li nastaven na log.1 zabezpečuje udržování řadíče msCAN12 ve stavu RESET a tím umožňuje nastavení některých řídících registrů, které není možné nastavit v normálním režimu obvodu. Mezi tyto registry patří:

- řídící registr CMCR1,
- registr časování obvodu CBTR0-1,
- řídící registr vstupních filtrů CIDAC,
- registry „šablon“ identifikátorů zpráv v příchozích filtrech CIDAR0-3 a
- registry masek CIDMR0-3.

CMCR1

Registr obsahuje velmi důležitý bit CLKSRC, který je-li nastaven na log. 1 zajišťuje připojení obvodu na vnitřní hodinový signál mikropočítače, který má dvojnásobnou frekvenci budícího krystalu procesoru. Informace o frekvenci taktování řadiče je nutná pro stanovení hodnot registrů časování celého obvodu a tím i pro volbu přenosové rychlosti.

Rád bych na tomto místě ještě upozornil na bit LOOPB, který podle návodu výrobce jenočipu je-li nastaven na hodnotu log. 1, umožňuje byužít tzv. „loop back mode“, což znamená přivedení výstupního toku dat zpět na vstup a tím dovoluje jednoduché ladění běžící aplikace.

Upozornnění: *Pokusil jsem se řadič v režimu loop-back použít, ale neúspěšně. V chybové dokumentaci k jednočipu 68HC12D60 dostupné na webowských stránkách firmy Motorola jsem našel informaci, že obvod z důvodu vnitřní chyby, nefunguje správně, a proto jej nelze v tomto režimu provozovat .*

4.5 Časovaní obvodu msCAN12

Časování obvodu vzhledem ke sběrnici CAN se provádí nastavením hodnot časovacích registrů CBTR0 a CBTR1. Pro stanovení hodnot jednotlivých parametrů je nutná znalost problematiky časování sběrnice CAN jak bylo popsáno v kapitole 2.

CBTR0 (bus timing register 0)
SJW0, SJW1 - udávají hodnotu tzv. „synchronization jump width“, což je maximální hodnota v časových kvantech o kterou může být čas určený pro jeden bit prodloužen popřípadě zkrácen. Hodnota obou bitů odpovídá počtu časových kvant.

BRP0-5 - hodnota tzv. „baud rate prescaler“ na jehož základě je stanovena doba časového kvanta T_q podle následujícího vztahu

$$T_q = \frac{Prescaler}{f_{CGMCANCLK}},$$

kde $Prescaler$ je hodnota bitů BRPx a $f_{CGMCANCLK}$ je frekvence hodinového signálu obvodu msCAN12 (v našem případě to byl dvojnásobek hodinové frekvence oscilátoru tedy 16MHz).

CBTR1(bus timing register 1)
SAMP - je-li hodnota tohoto bitu log. 1 znamená to, že vzorkování signálu je provedeno třikrát během doby jednoho bitu a jako výsledná hodnota je určena většinová úroveň.

TSEGxx (time segment)- hodnoty TSEG1x a TSEG2x určují pozici tzv. SAMPLE POINTU, tj. okamžiku kdy dochází ke vzorkování sběrnice. TSEG1x musí dle specifikace CAN nabývat hodnot od 1 do 16 časových kvant T_q a TSEG2x hodnot 1 až 8.

Mějme obvod, který chceme připojit k síti s danou přenosovou rychlostí a tedy potřebujeme určit hodnoty časovacích registrů CBTR1 a CBTR0.

- z přenosové rychlosti vypočteme periodu signálu $T_{per} = \frac{1}{speed_{bps}}$,
- určíme periodu signálu oscilátoru $T_{osc} = \frac{1}{f_{osc}}$,
- podle fyzikálních vlastností přenosového média zvolíme pozici vzorkovacího bodu a tím i hodnoty TSEG1x a TSEG2x a určíme celkový počet časových kvant připadajících na jeden bit a podle následujícího vztahu dobu nutnou pro přenesení jednoho bitu požadovanou rychlostí

$$T_{bit} = (SYNCSEG + TSEG1 + TSEG2)T_q,$$

- ze získaných hodnot určíme hodnotu „prescaleru“ dle následujícího vztahu

$$Presc. = \frac{T_{bit}}{T_{osc}}.$$

4.6 Přerušení

Obvod msCAN12 představuje pro procesor zdroj čtyř různých druhů přerušení.

- *receive interrupt* - přerušení od přijímacího bufferu,
- *transmit interrupt* - přerušení od jednoho ze tří vysílacích bufferů dojde-li k jejich vyprázdnění,
- *wake-up interrupt* - obvod může přejít v době neaktivity na lince do úsporného režimu a odposlouchávat provoz na sběrnici. Djde-li k opětovnému zahájení komunikace vygeneruje wake-up přerušení.
- *error interrupt* - chybové přerušení. Dojde-li na základě chybových čítačů k přechodu obvodu do jednoho z chybových režimů je vygenerováno příslušné přerušení. Výše uvedený text neobsahuje popis stavů obvodu a podmínek přechodu mezi jednotlivými stavy z důvodu omezeného rozsahu této diplomové práce. Proto doporučuji prostudovat [6].

Následuje výpis a základní popis řídících registrů souvisejících s přerušeními od obvodu msCAN12.

CRFLG - receive flag register
Registr se skládá z příznakových bitů signalizujících příčinu přerušení. Bity v tomto registru jsou pouze pro čtení. Zápis log. 1 na příslušnou pozici má za následek vynulování příslušného flagu. Rád bych zde upozornil na flag RXF na pozici 0, což je bit který signalizuje, že v zásobníku **receive buffer** se nacházejí nová úspěšně přijmutá data připravená pro další zpracování handlerem přerušení.

Bitům v registru CRFLG odpovídá bit v registru **CRIER** (Receive interrupt enable flag) sloužící k maskování příslušného přerušení. Je-li tedy hodnota bitu v registru CRIER odpovídající přerušení v reg. CRFLG nastavena na log. 0 k přerušení nedojde - bude ignorováno.

CTFLG - transmit flag register
Registr, který mimo jiné obsahuje bity TX2, TX1 a TX0, které identifikují zda je příslušný vysílací buffer prázdný a zda je tedy schopen přijmout od běžící aplikace nová data určená k odeslání, aniž by došlo k přepsání ještě neodeslaných informací.

CTCR - transmitter control register
Tento registr slouží jako řídící registr pro obsluhu vysílacích bufferů. Obsahuje bity TXEIE2-0, které umožňují maskovat přerušení od vyprázdnění odchozích bufferů,

4.7 Filtrování zpráv

Obvod msCAN12 je schopen díky vstupním filtrům „odstínit“ nežádoucí zprávy a díky tomu velmi výrazně snížit zatížení procesoru spojené s častými přerušeními způsobenými „nežádoucími“ příchozími zprávami.

Pro správné nastavení struktury vstupních filtrů je určen řídící registr CIDAC. Pro nastavení šablon zpráv, které budou akceptovány, slouží registry CIDAR0-7 které obsahují identifikátory přípustných zpráv. Pro zobecnění akceptovatelných identifikátorů jsou dále v obvodu implementovány registry masek CIDMR0-7, které obsahují informace o tom, které bity nás na příslušných pozicích identifikátoru nezajímají. Pro detailnější popis nastavení a použití vstupních filtrů bych rád čtenáře odkázal na [10]

4.8 Odeslání dat

V tomto odstavci bych rád popsal postup obsluhy obvodu při odeslání dat prostřednictvím obvodu msCAN12. Mějme tedy data ve formě nanejvýš osmi bajtů, pro představu postačující 11-bitový identifikátor a příslušně inicilizovaný obvod msCAN12(nastavena přenosová rychlost, registry namapovány na příslušná místa v paměti). Pro odeslání dat je nutno učinit následující:

- podle stavového registru CTFLG zjistit zda je nějaký odesílací registr volný a není-li čekat do doby než budou data odeslána.
- Uložit data na příslučné místo do paměti,
- uložit identifikátor a zajisti správné nastavení bitů RTR a IDE (v našem případě oba bity na log. 0),
- pokud to vyžaduje situace (není nutné) změnit hodnotu prioritního registru příslušného volného bufferu TBPRx

a zapsat log. 1 na příslušnou pozici bitu TXEx v registru CTFLG a tím odstartovat v nejbližším možném odeslání zprávy.

4.9 Příjem dat

V praktické části mé diplomové práce jsem řešil problém příjmu dat pomocí obsluhovací rutiny přerušení a zaregistrováním příslušného přerušovacího vektoru. Není to jediný způsob jak příjem ošetřit v aplikaci příjem dat. Lze například udělat nekonečnou smyčku a čekat na změnu bitu v registru CRFLG. Obsluha obvodu je obou případech shodná a proto bych postup rád uvedl.

- Předpokládejme, že je obvod zprávně nainicializován a že jsme povolili v registru CRIER přerušení od přijímacího bufferu nastavením RXFIE na log. 1 a úspěšně zaregistrovali příslušný vektor přerušení.
- Dojde-li tedy k přerušení, což znamená, že zpráva prošla vstupními filtry můžeme vyzvednout data z přijímacího bufferu a předat je aplikaci (např. přesunutím do vyrovnávací paměti FIFO),
- Máme-li tedy data uložena mimo příchozí buffer, musíme vynulovat RXF zápisem log. 1 na místo 0.bitu registru CRFLG.

Výše uvedené postupy jsem implementoval v praktické části diplomové práce a zdrojový kód je uveden v příloze A.

4.10 Fyzická vrstva

Vývody řadiče msCAN12 `CANH` a `CANL` jsou vyvedeny na dva výstupní piny pouzdra procesoru. V reálné aplikaci je nutné převést signály z řadiče na odpovídající fyzickou reprezentaci signálů pomocí galvanicky odděleného budiče tak jak bylo popsáno v kapitole 2.

5 Vývojové prostředí CodeWarrior a Metrowerks OSEKbuilder

5.1 CodeWarrior

Program CodeWarrior verze ADS 2.0 od firmy Metrowerks v licenci zakoupené pro Katedru řídící techniky umožňuje vývoj aplikací určených pro 8-mi, 16-ti bitové mikroprocesory Motorolla řady HC08 a HC12. Pro psaní zdrojového kódu aplikačního softwaru lze použít jak asembler příslušného procesoru, tak programovacího jazyka C podle normy ISO ANSI C. Prostředí CodeWarrior krom nástrojů pro psaní a editaci zdrojových kódů, obsahuje nástroj pro nahrávávní spustitelného programu do cílového systému a dále také velmi užitečný simulátor, který umožňuje velmi jednoduché ladění aplikačního software bez nutnosti nahrání do cílového sytému. Pro detailní popis prostředí a postupu při vývoji aplikace doporučuji [3] a [4].

5.2 Metrowerks turboOSEK

turboOSEK OS je softwarový produkt firmy Metrowerks, který je implementací operačního systému OSEK/VDX od firmy Metrowerks. Balík obsahuje knihovny operačního systému, generátor systému a grafický konfigurační nástroj OSEKbuilder, který umožňuje uživateli konfigurovat vyvíjenou aplikaci v grafickém uživatelském prostředí. Grafická reprezentace systémových objektů je uložena v jazyce OIL do konfiguračního souboru, podle kterého jsou generátorem operačního systému, který funguje jako modul programu CodeWarrior, vygenerovány jednotlivé objekty. Současná licence programu vlastněná katedrou K335 umožňuje vývoj aplikací pro 8-mi a 16-ti bitové mikroprocesory HC08 a HC12 od firmy Motorolla.

Implementace turboOSEK v současné licenci vlastněné K335 má následující parametry:

- určen pro cílové procesory HC08 a HC12,
- třídy konformity BCC1 a ECC1,
- komunikační třída konformity CCCA,
- až 2 časovače,
- softwarové i hardwarové čítače.

Tabulka 10 uvádí maximální možné počty a hodnoty související s objekty v systému.

Počet priorit úloh a systémových zdrojů	253
Počet úloh ve stavu `suspended`	253
Počet událostí na úlohu	32
Počet systémových zdrojů	255
Počet ISR	32
Počet jiných OSEK objektů	255

Tabulka 10: Maximální možné počty v turboOSEK.

5.3 Praktické zkušenosti s operačním systémem turboOSEK

V následující části dokumentu bych rád uvedl praktické zkušenosti, které jsem získal při psaní aplikace určené pro operační systém turboOSEK v nástroji Code Warrior a Osek Builder od firmy Metrowerks .

5.3.1 Založení nové aplikace

Postup jak založit a správně nastavit aplikaci, tak abychom dostali spustitelný soubor je velmi srozumitelně popsán v [3]. Rád bych se proto zaměřil na odlišnosti, které vznikají díky použití nestandardního způsobu nahrávání do FLASH paměti a spouštění aplikačního kódu na kartách CAN modulů.

Cílový systém (Target system)
Program CodeWarrior umožňuje připojení cílového systému na bázi procesoru 68HC12 pouze prostřednictvím tzv. BDM portu (viz.[6]), což je jednovodičové připojení 16-bitových mikropočítačů řady HC12, které je určeno pro komunikaci s cílovým procesorem tj. nahrávání aplikačního kódu do FLASH paměti, odlaďování a spouštění aplikace. Karty CAN modulů nejsou v současné době tímto rozhraním vybaveny, a proto nahrávání a spouštění aplikačních kódů probíhá prostřednictvím sériové linky tj, přes sériový port počítače.

Z výše uvedeného důvodu je při založení aplikace jako cílový (target) systém nutno vybrat Simulátor a ne konkrétní klon mikropočítače HC12.

Změna mapování paměti
Po automatickém vygenerování souborů a adresářů nového projektu, je nezbytné změnit rozsahy jednotlivých segmentů adresovém prosotru, které budou přiděleny pro zásobník (STACK), operační paměť (RAM), paměť programu (ROM) a pro vektory přerušení. Toto nastavení je uvedeno v souboru `default.prm` v části SECTIONS a PLACEMENT kde jsou uvedeny rozsahy adres a mapování jednotlivývh částí do paměti. Doporučuji změnit obsah souboru následovně:

```
SECTIONS
    MY_RAM = READ_WRITE  0x0200 TO 0x04FF; //operační paměť
    MY_STK = READ_WRITE  0x0500 TO 0x07FF; //zásobník
    MY_ROM = READ_ONLY   0x8000 TO 0xFF7F; //paměť programu
    VECTORS= READ_ONLY  0xFFC2 TO 0xFFFF; //vektory přerušení

PLACEMENT  // mapování do paměti
    DEFAULT_ROM  INTO  MY_ROM;
    DEFAULT_RAM  INTO  MY_RAM;
    .stackstart  INTO  MY_RAM;
    .stack       INTO  MY_RAM;
    .stackend    INTO  MY_RAM;
    .vectors     INTO  VECTORS;
END

STACKSIZE  0x200 // velikost zásobníku v bytech
```

V případě, že při kompilaci program CodeWarrior oznámí nedostatek paměti nebo přetečení zásobníku, doporučuji změnit velikost zásobníku a jeho rozsah v části SECTIONS.

Nastavení linkeru

Vzhledem k tomu, že softwareový nástroj `VLoader` napsaný Miroslavem Musilem (viz. [7]), určený pro nahrávání binárních souborů do FLASH paměti přípravku pracuje pouze se soubory s koncovkou `*.s19` a linker programu CodeWarrior defaultně kompiluje binární soubory do formátu `*.hex`, je nutné změnit nastavení linkeru. Přenastavení lze provést v menu Edit→Generic (Simulator) Settings→ v poli Target Settings Panel kliknout na položku Linker for HC12 → tlačítko Options *to* okno Smart Linker Option Settings *to* záložka Output *to* zaškrtnout položku Generate S-record file *to* potvrdit Apply a OK. .... zaškrtnutím volby ... srecord file . Výstupní spustitelný binární soubor bude po kompilaci uložen v adresáři `bin` umístěném v kořenovém adresáři projektu a bude mít koncovku `*.sx`.

Změna OIL souboru

Vzhledem k tomu, že jsme jako cílový systém vybrali Simulator, bude v definičním souboru `*.oil` v objektu `OS` nastaven parametr `TargetMCU` tj. cílový systém, na defaultní hodnotu. V našem případě změníme jeho hodnotu následovně

```
TargetMCU = HC12D60 {
};
```

což je mikropočítač použitý v přípravcích CAN modulů. Ponecháme-li defaultní nastavení, dojde při případné obsluze přerušení k deadlocku celého systému.

5.3.2 ISR - obslužná rutina přerušení

Chceme-li aby v OSEK aplikaci došlo k obsoužení maskovatelného přerušení je nutné napsat a zaregistrovat obslužnou rutinu přerušení ISR. Doporučuji postupovat podle následujícího postupu:

- v programu Osek Builder vložit objekt `ISR`, pojmenovat jej a nadefinovat příslušné parametry tj. prioritu `PRIORITY` a příslušnou kategorii `CATEGORY` jak bylo popsáno v kapitole 3.9.1,

  ```
  ISR msCANISR {
     CATEGORY 3; // 1..3
     PRIORITY 0;
     }
  ```

- V souboru `vector.c` zaregistrovat identifikátor `msCANISR` pomocí služby `DeclareISR` a nahradit vektor `dummyISR` u příslušného vektoru přerušení identifikátorem uživatelské rutiny,

  ```
  ...
  DeclareISR(msCANISR);
  ...
  OSVECTF dummyISR;
  OSVECTF msCANISR; // 0xFFCF msCAN receive vector
  OSVECTF dummyISR;
  ...
  ```

- do souboru `main.c` vložit tělo obslužné rutiny.

  ```
  ISR msCANISR {
  /*tělo obslužné rutiny*/
  }
  ```

použijeme-li standardní konstrukce handleru obvyklé v programu CodeWarrior

```
interrupt 28 void handler(void){
...
}
```

dojde z pohledu operačního systému OSEK k zaregistrování **ISR kategorie 1**, tj. v těle obslužné funkce nebude možné používat služeb operačního systému.

Pozn.: *V případě, že se nepodaří zkompilovat zdrojové soubory a překladač hlásí chybu, že není definován identifikátor ISR, doporučuji zkontrolovat výše uvedný postup (především definici těla obslužné rutiny v souboru main.c a popřípadě restartovat program OSEKbuilder.*

5.3.3 EVENTs - události

Události v operačním systému OSEK slouží k synchronizaci úloh a k předávání binárních informací. Oproti jiným operačním systémům (např. LINUX RTAI) částečně zastupují funkci tzv. binárních semaforů. Pro správnou funkci a především úspěšné zkompilování zdrojových kódů je nutno učinit následující kroky:

- v souboru `*.OIL` nadefinovat objekt EVENT

  ```
  EVENT DataReady {MASK AUTO},
  ```

  kde `DataReady` je identifikátor právě definované události,

- dále přiřadit události jejího „majitele“,

  ```
  TASK Task1 {
    ...
    EVENT DataReady;
    STACKSIZE 50; // parametr AUTO nefunguje
    TYPE EXTENDED; // pouze rozšířená úloha smí čekat na událost
    ...
  };
  ```

- v deklarační části souboru `main.c` použít službu operačního systému

  ```
  DeclareEvent(DataReady);
  ```

- a v těle jednotlivých úloh[4] používat služby jako např..

  ```
  SetEvent(Task1,DataReady);  // udalost nastala
  GetEvent(DataReady); // cekani na udalost
  ClearEvent(DataReady); // deaktivace události
  ```

[4]Službu ClearEvent lze použít pouze v těle „majitele“ události, tj. v těle úlohy Task1

6 Použitý hardware - CAN moduly

CAN moduly jsou elektronické karty vyvinuté v předchozí diplomové práci Miroslavem Musilem, pracující na bázi mikroprocesoru Motorolla HC12D60, který je vybaven řadičem sběrnice CAN mcCAN12 jehož obsluha a použití bylo posáno v odstavci 4. Jsou určeny pro výuku v laboratoři Katedry řídící techniky ČVUT FEL v Praze, která v současné době vlastní tři exempláře, umožňující výuku a experimety při použití sběrnice CAN. Karty jsou osazeny obvody budičů Philips PCA82C50 pro buzení fyzické vrstvy CAN Hi-speed, které umožňují vzájemnou komunikaci po síti CAN. Pro detailnější popis CAN modulů si dovoluji odkázat na [7].

7 Komunikační vrstva pro OS OSEK - praktická část

Jak již bylo posáno v kapitole 3, implementace operačního systému turboOSEK od firmy Metrowerks v zakoupené licenci obsahuje pouze komunikační třídu konformity CCC1A. V praxi to znamená, že turboOSEK dovoluje psát aplikační software, který používá komunikační služby pouze vrámci jednoho procesoru a neumožňuje psát distribuované aplikace komunikující prostřednictvím sběrnice CAN. Z tohoto důvodu jsem implementoval základní komunikační funkce pro inicializaci řadiče msCAN12 `initCOM`a pro odeslání a příjem standardních rámců s 11-bitovými identifikátory `SendMsg` a `Received`.

7.1 Inicializace

Pro správnou činnost systému je nutné inicializovat řadič sběrnice CAN msCAN12 za použití řídících registrů popsaných v odstavci 4.4. K tomu slouží funkce

```
void InitCOM(void);
```

Funkce volá rutinu incializace řadiče msCAN12 `initMSCAN` z knihovny `msCAN.h`, která namapuje přijímací a odesílací zásobníky a řídící registry do paměti, nastaví jejich hodnotu a pomocí funkce `_initTiming()` nastaví časování řadiče. Funkce je součástí knihovny `com.h`.

7.2 Komunikační funkce

7.2.1 Odeslání dat

K odeslální dat je určena funkce `SendMsg()` z knihovny `com.h` s následující syntaxí:

```
unsigned char SendMsg(int *id,char* data,unsigned char* length);
```

kde

- `id` - ukazatel na 11-bitový identifikátor rámce,
- `data` - ukazatel na pozici pole dat. Je-li jeho hodnota `NULL` je přenášený rámec vyslán jako remote rámec.
- `length` - počet přenášených dat v bajtech (0..8).

Funkce vrací počet odeslaných bytů dat.

7.2.2 Příjem dat

příjem dat lze uskutečnit dvěmi způsoby:

- pomocí přerušení,
- pomocí nekonečné smyčky, ve které bude testováno jestli nepřišla nová data.

Jsou-li přijata nová data je nutno v obou případech vyjmou přízhozí zprávu z přijímacího zásobníku a povolit další přerušení popř. příchod nových dat. Napsal jsem proto univerzální funkci `Received(..)`, která vyjme data ze zásobníku a povolí příjem dalších dat. Funkci lze použít jak v obslužné rutině přerušení, tak v nekonečné čekací smyčce s následující syntaxí:

```
unsigned char Received (int *id, char *data, char * length);
```

kde

- `id` - ukazatel na proměnnou kde má být uložen identifikátor,
- `data` - ukazatel na pole kam mají být uložena příchozí data,
- `length` - ukazatel na proměnnou kam má být uložena informace o počtu přijatých bajtů.

Funkce vrací informaci o počtu přijatých dat. V případě příjmu remote rámce, je vrácena hodnota 0 a ukazatel data je natsaven na hodnotu `NULL`. Funkce je součástí knihovny `com.h`.

8 Řídící jednotka automatické klimatizace

V dnešní době si již jen těžko dovedeme představit moderní automobil bez klimatizační jednotky udržující stálou uživatelem nastavenou teplotu v kabině vozu. V letním období, kdy venkovní teploty ve stínu často převyšují třicet stupů celsia se prosklená kabina vozu stává nebezpečnou výhní jejíž teploty mohou dosahovat až k hodnotě sta stupňů celsia. Je prokázáno, že lidský organismus není při dlouhodobém pobytu schopen takovýmto teplotám odolávat a z tohoto důvodu není řidič schopen udržovat pozornost nezbytnou pro řízení motorového vozidla a účasti na silničním provozu. Klimatizační jednotka má proto kromně nezanedbatelného vlivu na zvýšení komfortu z cestování také vliv na bezpečnost přepravovaných osob. Jejím úkolem je tedy udržovat teplotu v prostoru kabiny na přijatelné úrovni v okolí uživatelem nastavené hodnoty a to jak v období kdy je venkovní teplota vyšší než stanovená ale i v období kdy venkovní teplota klesá pod bod mrazu a je tedy nutno kabinu vytápět.

8.1 Popis zařízení

Řízení teploty v prostoru pro cestující je prováděno pomocí změny teploty nasávaného venkovního vzduchu jehož vnitřní energie a tedy i teplota je ovlivňována pomocí tepelného výměníku, kterým vzduch nuceně (náporově nebo díky ventilátoru) prochází. Tepelný výměník tedy musí být schopen energii jak dodávat tak odebírat.

Je-li nutno dodávat energii (systém topí) je ve valné většině vyráběných systémů k této změně využito „odpadní teplo“ vznikající při činnosti spalovacího motoru vozidla a pomocí změny úhlu natočení trojcestného ventilu (akční člen) je regulován výkon topné části klimatizační jednotky (viz. obrázek 18) a tedy i teplota vzduchu vstupujícího do kabiny.

Je-li nutno energii z prostoru pro cestující odebrat používá se k tomuto účelu venkovní vzduch, který prošel chladícím zařízením. Snížení teploty venkovního nasávaného vzduchu je

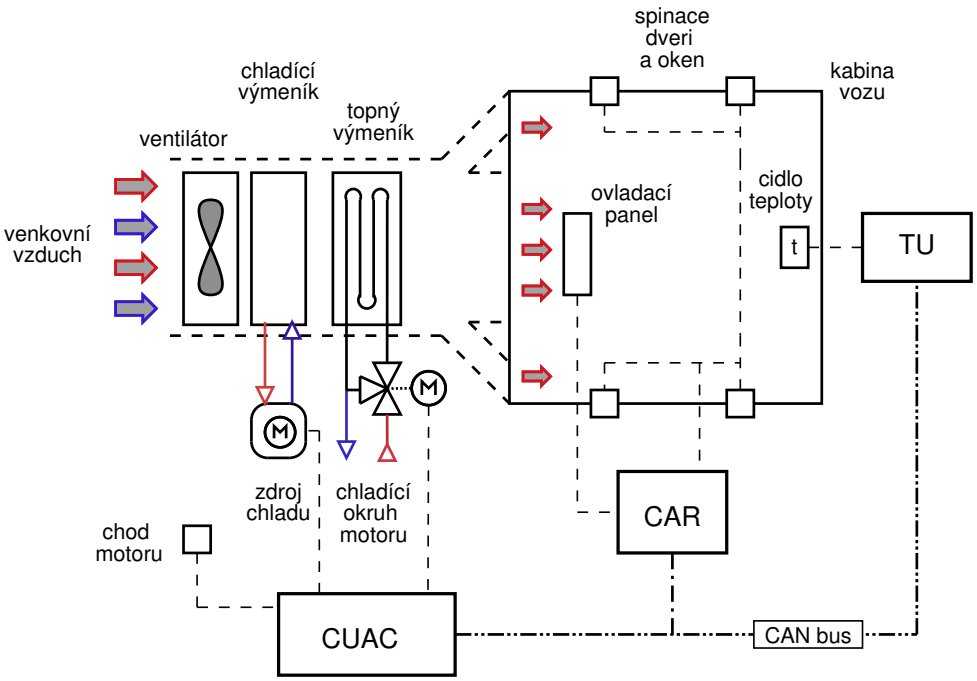
podobně jako u topné části provedeno pomocí průchodu vzduchu chladícím výměníkem umístěným v cestě nasávanému vzduchu. Chladící výměník je ve skutečnosti výparníkem chladícího systému.

V principu se jedná o uzavřený systém, který je rozdělen na vysokotlakou a nízkotlakou část. Ústředním výkonným členem je kompresor, který stlačuje chladící médium. Při stlačení dojde ke zvýšení teploty plynu, což vyplývá ze stavové rovnice plynu a je tedy nutno zvýšenou energii vhodným způsobem odebrat. Mezi vysokotlakou a nízkotlakou částí je umístěna tryska, kterou stlačené médium prochází, expanduje ve výparníku a prudce se ochlazuje. Výparník je pak umístěn v cestě nasávanému venkovnímu vzduchu a ovlivňuje tak jeho vnitřní energii. Pro detailní popis principu chladícího systému doporučuji [11]

Dále vycházejme z přepokladu, že výkonná část chladící jednotky není poháněna mechanicky přímo z motoru, ale je poháněna elektrickou energií. Vzhledem k energetickým nárokům, které v takovém případě klade chladící zařízení na akumulátor a alternátor vozu, není možné klimatizační jednotku provozovat, není-li v chodu motor vozidla, tedy dostatečný zdroj energie a jsou-li otevřena okna nebo dveře automobilu, tedy příliš velká ztráta chladného vzduchu. Nebyla-li by zajištěna vzájemná vazba mezi řídícím algoritmem a stavem vozu, mohlo by na jedné straně dojít k poškození chladící jednotky a na straně druhé k destrukci nebo k vážnému poškození elektrických zařízení vozu.

8.2 Technologické schéma

Na obrázku 18 je zobrazeno předpokládané technologické schéma jednotky automatické klimatizace. Ventilátorem nasávaný venkovní vzduch prostupuje nejprve přes chladicí a pak přes ohřívací výměník do prostoru pro cestující. Teplota v kabině vozu (v prvé horní části obrázku) je měřena pomocí „dálkově ovládaného“ teplotního čidla (karta TU) a přenášena po sběrnici CAN do řídící jednotky (karta CUAC), kde je dále zpracována.



Obrázek 18: Technologické schéma klimatizační jednotky.

8.3 Realizace řídící jednotky automatické klimatizace

Pro realizaci výše popsaného systému řízení automatické klimatizace a pro simulaci stavu automobilu (stav dveří, oken, motoru) jsem použil tři exempláře CAN modulů vybavených mikrokontrolérem Motorola 68HC12D60.

Karta CUAC je výkonnou částí systému automatického řízení teploty, karta TU představuje inteligentní dálkově ovládané teplotní čidlo a karta CAR simuluje interakci s uživatelelem a informace o stavech jednotlivých zařízení na něž je chod řídící jednotky vázán.

8.3.1 Karta CUAC

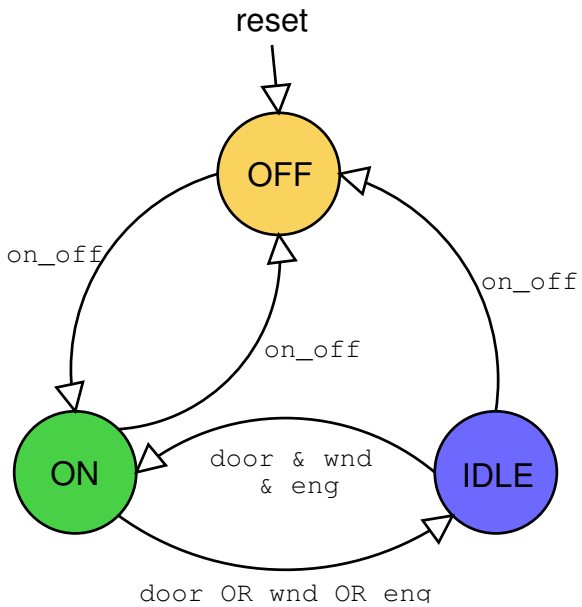
Na kartě CUAC jsem implementoval algoritmus přechodu mezi jednotlivými stavy a zároveň algoritmus třípolohové regulace teploty prostoru.

Stavový diagram řídící části

Řídící jednotka se může nacházet v jednom ze tří stavů:

- OFF - vypnuto.
- ON - režim řízení teploty,
- IDLE - útlumový režim.

Obrázek 19 zobrazuje stavy řídící jednotky a podmínky přechodu mezi nimi.



Obrázek 19: Stavový diagram řídící jednotky klimatizace.

Po resetu přejde jednotka do stavu OFF kde zůstává do doby kdy je uživatelem vyžádána její aktivace pomocí tlačítka ON/OFF a po té přejde do stavu ON, tedy režimu normálního chodu. Ve stavu ON je aktivní dvoupolohový regulátor, na základě regulační odchylky e mění stav digitálních výstupů `heating` a `cooling` a tím ovládá pohon trojcestnoho ventilu a spíná chladící zařízení. Tímto způsobem řídí teplotu T_{in} v prostoru pro cestující.

Dojde-li k vypnutí motoru a je-li jednotka ve stavu ON, přejde celé zařízení do stavu IDLE. Jadnotka přejde do stavu IDLE také případě dojde-li k otevření alespoň jednoho okna (koncový spinač) nebo dojde-li k otevření dveří automobilu. Ve stavu IDLE nelze měnit žádanou hodnotu teploty T_w a je odstaven třípolohový regulátor pro řízení akčních členů (obě akční veličiny se rovnají nule).

Ze stavu IDLE jednotka přejde do stavu ON v případě, že jsou zavřena okna i dveře automobilu a běží-li motor. DO stavu OFF jednotka přejde na základě požadavku uživatele tj. je-li použito tlačítko zapnutí/vypnutí klimatizace.

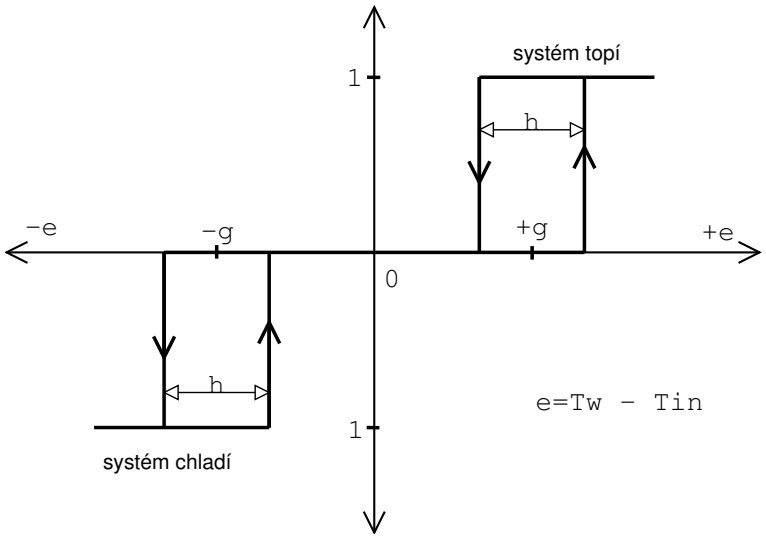
Regulace teploty - popis algoritmu

Na kartě CUAC jsem implementoval dvoupolohový regulátor se dvěma výstupy. Výstup `heating` slouží k ovládání pohonu trojcestného ventilu a výstup `cooling` je určen k ovládání chodu chladícího zařízení.

Podle znaménka regulační odchylky spočtené podle následující rovnice

$$e = T_w - T_{in},$$

kde T_w je žádaná hodnota teploty a T_in je skutečná teplota v prostoru pro cestující, nastaví regulátor digitální výstup `cooling` (v případě je-li $e < -g$)nebo `heating` (je-li $e > +g$)na úroveň log. 1 (viz obrázek 20). Je-li rozdíl e v pásmu necitlivosti regulátoru g oba dig. výstupy mají hodnotu log. 0.



Obrázek 20: Regulační křivka regulátoru.

Popis řídícího programu

Po resetu jednotky je spuštěna úloha `InitTask`, která

- uložením hodnoty 0x00 do registru COPCTL odstaví watchdog mikročipu,
- namapuje, vynuluje a nastaví port PORTP jako výstupní,

- inicializuje funkcí `InitCOM` řadič msCAN,
- nastaví cyklický alarm `CyclicAlarm`,
- nastaví hodnotu globálních proměnných stavu
- aktivuje úlohy SendStateTask a SignalTask a je pomocí služby `TerminateTask` ukončena.

Periodický alarm `CyclicALarm` způsobí opakované spouštění řídící úlohy `CyclicTask`, která na základě promenné `state` reprezntující funkční stav jednotky provádí řídící operace.

- `state==ON` - je spuštěna řídící funkce `Control`, která obsahuje implementaci regulátoru popsanou v předchozím odstavci. Na základě návratové hodnoty funkce `Control` je nastaven příslušný výstup `cooling` nebo `heating`. Došlo-li k otevření dvěří nebo oken, nebo k vypnutí motoru přechází jednotka do stavu IDLE.
- `state==OFF` - jednotka vypnuta. Oba výstupy jsou nastaveny na hodnotu log 0.
- `state==IDLE` - mód nečinnosti. Oba výstupy jsou nastaveny na hodnotu log. 0. Podle stavu proměnných `door, window, engine` je posouzen případný přechod do stavu ON.

Před ukočením úlohy jsou aktivány události `SendState`, která způsobí aktivaci úlohy `SendState` a tím vyslání stavového bytu a událost `SignalEvent`, která umožní aktivaci úlohy `SignalTask` zobrazující stav vnitřních proměnných na port `PORTH`.

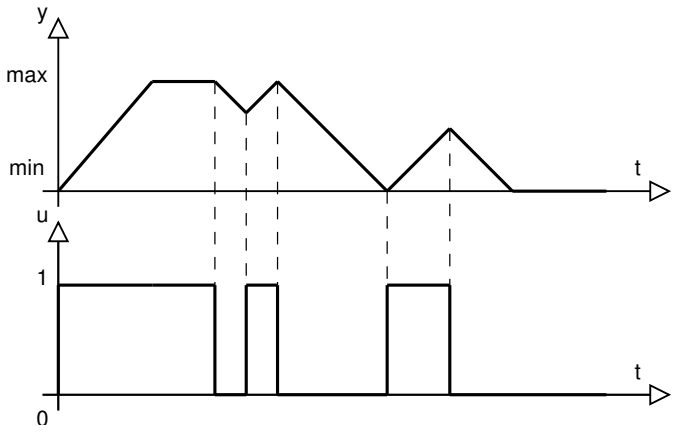
Obslužná rutina přerušení `MSCANreceive` podle přijmutého identifikátoru mění hodnoty globálních proměnných a tím ovlivňuje stav řídící jednotky.

Výpis kódu řídící aplikace je na přiloženém CD v adresáři `/CCU/CUAC/Sources`.

Pohon trojcestného ventilu

Z výše uvedeného algoritmu plynou požadavky kladené na pohon trojcestného ventilu umístěného v topném okruhu klimatizační jednotky. Vzhledem

k tomu, že používáme dvoupolohovou regulaci je nutné aby byl pohon integračního charakteru, což znamená aby v případě log. 1 na vstupu, pohon otevírál ventil konstantní rychlostí až do maximálního úhlu otevření a v případě log. 0 konstaní rychlostí zavíral až do zkratování tepelného výměníku (viz obrázek 21).



Obrázek 21: Úhel otevření ventilu y v závislosti na hodnotě řízení u.

8.3.2 Karta CAR

Interakci s uživatelem a návaznost na stav podstatných částí automobilu jsem realizoval pomocí karty CAR. Kartu jsem dovybavil jednotkou digitálních vstupů a výstupů. Dig. vstupy jsou připojeny na bránu portu P (schéma zapojení je uvedeno v příloze B) a dig. výstupy jsou připojeny na bránu portu G. Po stisku tlačítka je pomocí funkce `SendMsg` vyslána zpráva obsahující příslušný identifikátor a v datovém poli kopii stavu portu digitálních vstupů P.

Tabulka 11 uvádí symbolický význam jednotlivých digitálních vstupů.

jméno	bit	popis
`ONOFF`	P.0	tlačítko ovládání chodu klimatizační jednotky
`ENG`	P.1	simulace chodu motoru automobilu
`DOOR`	P.2	simulace spínače dveří
`WND`	P.3	simulace koncového spínače oken
`PLUS`	P.4	změna žádané hodnoty o $+0.5^oC$
`MINUS`	P.5	změna žádané hodnoty -0.5^oC
`WND_UP`	P.6	tlačítko ovládání l.p.okna vozu Škoda Fabia
`WND_DOWN`	P.7	tlačítko ovládání l.p.okna vozu Škoda Fabia

Tabulka 11: Symbolický význam digitálních vstupů karty CAR.

Popis programu

Po resetu je aktiována úloha `InitTask`, která provede pomocí služby `InitOS()` inicializaci operačního systému OSEK, dále použitím funkce `initCOM()`provede incializaci řadiče msCAN12 a nastaví pomocí služby `SetTimer()` alarm, tak aby priodicky aktivoval úlohu `ScanTask`. Po inicializaci portů P a G je úloha `InitTask`ukončena. Úloha `ScanTask` je periodocká úloha aktivovaná na základě impulsů z časovače `SystemTimer`. Po aktivaci porovná hodnotu portu P s hodnotu z předchozího cyklu a je-li odlišná určí, které tlačítko je právě zmáčknuto a po té pomocí funkce `SendMsg(...)` odešle zprávu s identifikátorem příslušné události a daty obsahujícími stav portu P. Po té je úloha deaktivována a celý cyklus se opakuje znovu. Výpis kódu řídící aplikace je k nahlédnutí na přiloženém CD v adresáři `/CCU/CAR/Sources`.

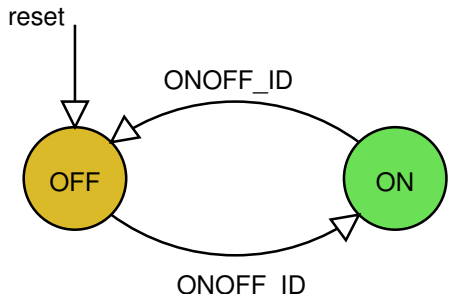
8.3.3 Karta TU

Karta TU plní v systému funkci dálkově ovládaného čidla teploty. Jak ukazuje obrázek 22, pracuje ve dvou režimech:

- ON - režim chodu a
- FF - režim nečinnosti.

Mezi stavy ON a OFF zařízení přechází po příjmu zprávy obsahující identifikátor `ONOFF_ID` vyslané řídící jednotkou CUAC na základě zprávy informující řídící jednotku o požadavku uživatele na vypnutí celého zařízení automatické klimatizace.

V režimu ON každých periodicky karta měří pomocí digitálně analogového převodníku, hodnotu napětí teplotního senzoru úměrnou měřené teplotě. Při realizaci jsem z demonstračních důvodů použil místo aktivního čidla teploty potenciometr. Teplota je tedy pomocí analogově digitálního převodníku převedena na hodnotu osmibitového čísla a prostřednictvím řadiče msCAN12 jako data odeslána s identifikátorem `TEMP_ID` po sběrnici CAN do řídící jednotky CUAC, která ji dále použije pro řízení.



Obrázek 22: Stavový diagram inteligentního teplotního čidla.

Je-li program v režimu OFF je pozastaveno cyklické měření teploty a nejsou vysílány žádné zprávy na sběrnici CAN. Přijde-li zpráva obsahující identifikátor `ONOFF_ID`, dálkově ovládané čidlo přejde do režimu ON posaného v předchozím ostavci.

Popis programu

Po resetu karty je spuštěna úloha `InitTask`, která vykoná inicializaci operačního systému (služba `InitOS`), provede inicializaci komunikační vrstvy (funkce `InitCOM`) a nakonec nastaví systémový časovač tak aby cyklicky spouštěl úlohu `ScanTask`. Nastane-li alarm `CyclicAlarm`, je na jeho základě spuštěna úloha `ScanTask`, která na základě proměnné `state`, jejíž hodnota je závislá na stavu jednotky, buď ukončí úlohu, nebo aktivuje analogově digitální převodník a odešle hodnotu s identifikátorem `TEMP_ID` do řídící jednotky. Po té je běh úlohy ukončen.

Úloha `MSCANreceive` je spouštěna na základě přerušení od příjmu zprávy řadičem ms-CAN12. Její úlohou je v tomto případě po příjmu zprávy s identifikátorem `ONOFF_ID` na základě přechozí hodnoty proměnné `state` přechod celého zařízení do opačného stavu (tj. změna hodnoty proměnné `state`). Výpis kódu řídící aplikace je k nahlédnutí na přiloženém CD v adresáři `/CCU/TU/Sources`.

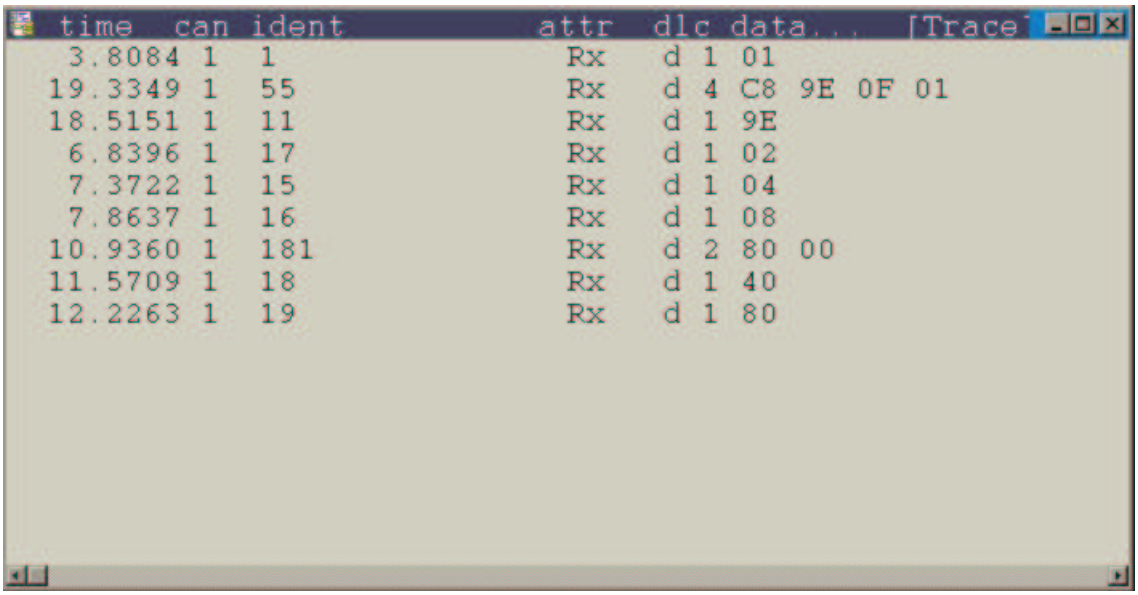
8.3.4 Přehled identifikátorů zpráv

Pro lepší přehlednost ještě závěrem uvádím tabulku všech identifikátorů zpráv, které jsou použity v systému řízení automatické klimatizace. Sloupce d0 až d4 představují obsah přenášeného datového pole zprávy.

identifikátor	hodnota	popis	d0	d1	d2	d3
`ID_TEMP`	`0x17`	identifikátor paketu přenášejícího okamžitou teplotu prostoru T_{in}	T_{in}	–	–	–
`ID_PLUS`	`0x11`	změna žádané hodnoty T_w	–	–	–	–
`ID_MINUS`	`0x11`	změna žádané hodnoty T_w	–	–	–	–
`ID_ON`	`0x11`	zpráva ovládání chodu karty CUAC a TU	–	–	–	–
`ID_STATUS`	`0x11`	identifikátor stavové zprávy	T_w	T_{in}	`PORTH`	`Control()`
`ID_DOOR`	`0x11`	identifikátor zprávy o změně stavu dveří	`PORTH`	–	–	–
`ID_WINDOW`	`0x11`	identifikátor zprávy o změně stavu oken,	`PORTH`	–	–	–
`ID_ENGINE`	`0x11`	identifikátor zprávy o změně stavu motoru	–	–	–	–

8.3.5 Výpis z CAN analyzátoru

Ke sběrnici CAN, po které komunikují jednotlivé karty CAN-modulů, jsem připojil CAN analyzátor. Výpis přenášených zpráv zachycených analyzátorem je uveden na obrázku 23. Po přivedení napájení, není komunikační sběrnice využívána. Po stisku tlačítka ON/OFF připojeného na digitální vstupy karty CAR je kartou CAR vyslána zpráva s identifikátorem `ID_ON`, která má za následek přechod řídícího algoritmu na kartš CUAC do stavu ON a přechod dálkově ovládaného čidla do stavu ON. Řídící jednotka CUAC ve stavu ON a IDLE periodicky vysílá zprávu s identifikátorem `ID_STATE` obsahující stavové informace. Jednotka TU vyzílá periodicky zprávu s identifikátorem `ID_TEMP`, která obsahuje aktuální změřenou hodnotu teploty prostoru kokpitu. Dojde-li ke změně stavu dveřních nebo okenních spínačů a nebo ke změně stavu motoru, je o tom kartou CAR vyslána příslučná zpráva. Hodnoty identifikátorů a obsah jednotlivých zpráv jsou uvedeny v předchozí tabulce.



Obrázek 23: CCU - výpis z CAN analyzátoru

9 Závěr

Cílem této diplomové práce bylo seznámit se s operačním systémem OSEK a s komunikačním protokolem CAN, dokončit na základě podkladů z předchozí diplomové práce [7], výrobu technických prostředků určených pro výuku v laboratoři Katedry řídicí techniky ČVUT FEL a na příkladu distribuované aplikace ověřit jejich funkčnost.

Operační systém OSEK poskytuje dostatečné mechanismy a služby pro běh řídicích aplikací a to díky možnosti optimalizace funkčnosti výkonného jádra pro platformy disponující malou operační pamětí. Řídicí aplikace může obsahovat mechanismy jako např. systém událostí, sdílené systémové zdroje, systém hardwarově vázaných alarmů atd. Oproti jiným operačním systémům v OS OSEK není defiován mechanismus synchronizace pomocí semaforů, který je však alespoň co se týče binárních semaforů, možno suplovat použitím sdílených systémových zdrojů.

Komunikační protokol CAN je díky vysoké míře zabezpečení přenášených informací ideálním prostředkem pro komunikaci v systémech reálného času. Používané fyzické vrstvy jsou koncipovány s ohledem na maximální odolnost proti elektromagnetickému rušení a ve verzi „CAN fault toleran“ dovolují komunikaci dokonce po poškozeném vedení. Protokol je zabezpečen proti chybám vzniklým při přenosu zprávy použitím kódování bit-stuffing, systémem pro dlouhodobé sledování chybových stavů zařízení a použitím CRC a to při přenosových rychlostech až 0,5 megabitů za sekundu (CAN hi-speed). Přístup uzlů na sériovou sběrnici je řešen díky vhodně zvolené reprezentaci logických reprezentací signálu pomocí mechanismu nazývaného „arbitráž“ popsaného v kapitole 2. Komunikace po síti je typu „broadcast - všichni slyší všechno“. Síťové uzly neobsahují adresy a ani informace o okolních zařízeních připojených do sítě což umožňuje jednoduché připojení nového zařízení a tím snadné rozšíření sítě.

Implementace operačního systému turboOSEK od firmy Metrowerks, kterou jsem měl k dispozici k vývoji ukázkové aplikace neumožňuje v licenci, kterou vlastní K335, meziprocesorovou komunikaci. V praktické části diplomové práce jsem proto krom dokončení výroby elektronických karet a vývoje a implementace distribuované aplikace nastudoval obsluhu a inicializaci obvodu řadiče sběrnice msCAN12(viz kap. 4) a naprogramoval jsem knihovnu umožňující přenos zpráv po sběrnici CAN a tím i komunikaci mezi řídicími jednotkami.

Program OSEKbuilder umožňuje jednoduché nastavení objektů použitých v OSEK aplikaci. V přehledném grafickém prostředí lze měnit parametry použitých objektů a jednoduše přidávat nebo ubírat nové komponenty. Výsledná konfigurace je uložena do OIL souboru, který je použit ve fázi generování systému OSEK. Program zjednodušuje proces tvorby OIL souboru a výrazně tím usnadňuje tvorbu řídicí aplikace.

V praktické části DP jsem navrhl a v programu CodeWarrior a OSEKbuilder implementoval algoritmus řízení klimatizace automobilu. Výsledná distribuovaná aplikace je rozdělena do tří částí. Modul CUAC představuje ustřední výkonnou část obsahující řídicí algoritmus popsaný v kap 8. Modul CAR simuluje stav zařízení automobilu, ne které je chod řídicí jednotky vázán a zároveň představuje vzdálený panel pro ovládání jednotky. Modul TU je naprogramován jako vzdálené, dálkově ovládané teplotní čídlo. Všechny tři moduly komunikují prostřednictvím sběrnice CAN.

V rámci praktické části diplomové práce jsem dále zajišťoval dokončení výroby elektronických karet navržených v předchozí diplomové práci Miroslavem Musilem. Zařízení pracují na bázi 16-ti bitového mikropočítače Motorola HC12D60, který je krom jiných periferií vybaven řadičem sběrnice CAN msCAN12 umožujícím komunikaci po sběrnici CAN. Karty jsou osazeny obvody pro buzení fyzické vrstvy CAN ve verzi „high-speed CAN“ PCA82C250 od firmy Philips.

V závěrečné fázi diplomové práce jsem uskutečnil experiment, kdy jsem se pokusil připojit moduly na sběrnici CAN vozu Škoda Fabia. Cílem tohoto experimentu bylo pokusit se ovlivnit chod řídicího algoritmu stavem některého ze zařízení vozu a zároveň, snaha ovládat zařízení vozu. V experimentu se mi podařilo ovládat levé přední okno a na základě stavu dvěří vozu, změnit stav řídicí jednotky klimatizace.

Poděkování

V závěru bych rád poděkoval panu Dr.Ing. Zdeňkovi Hanzálkovi za pomoc a vedení v průběhu tvorby této diplomové práce.

Jmenovitě bych rád poděkoval:

- *Ing. Pavlu Růžičkovi - za cenné rady při realizaci zařízení CAN modulů,*
- *Ing. Pavlu Píšovi - za trpělivost,*
- *Ing. Janu Krákorovi - za cenné rady při vývoji dokumentace DP,*
- *Zdeňku Soukupovi - za pomoc při osazování desek plošných spojů CAN modulů a*
- *Ing.Ivanu Krákorovi - za rady a konzultace týkající se klimatizačních zařízení.*

Dále bych rád poděkoval všem mým blízkým, kteří mne po celou dobu studia ČVUT FEL podporovali.

Reference

[1] *OSEK Communication Version 2.2.2*, OSEK Group,18.12.2000.

[2] *OSEK/VDX Operating System Version 2.1*, OSEK Group, 13.11.2000.

[3] *OSEK builder v. 2.2.1, user's manual*,Metrowerks, 2001.

[4] *CodeWarrior user's manual*,Metrowerks,2001.

[5] *OSEK trubo OS/12 v.2.1.1, technical reference*,Metrowerks, 2001.

[6] *68HC(9)12D60 Advanced information*,rev. 3, Motorola, 2001.

[7] *Diplomová práce - Miroslav Musil*,ČVUT Praha, 2003.

[8] *CAN specification 2.0, part A*, CAN in Automation.

[9] *CAN specification 2.0, part B*, CAN in Automation.

[10] *MSCAN interrupts - AN2283/D*,Motorola,2001.

[11] *Chladici a klimatizační systémy*, skripta ČVUT FSI, Vydavatelství ČVUT Praha, 1990.

[12] *Učebnice jazyka C*, Pavel Herout,nakladatelství Kopp 1997.

[13] *Na příliš stručný úvod do systému LaTeX 2ε*, Tobias Oetiker, Michal Kočer, Pavel Sýkora, leden 1996.

Odkazy na webovské stránky

- **CAN in automation**, www.cia.org,
- **OSEK/VDX**, www.osek.org,
- **Motorola**, www.motorola.com,
- **Katedra řídicí techniky ČVUT FEL**, dce.felk.cvut.cz.

10 Příloha A

10.1 Knihovna COM - výpis programového kódu

com.h

```
#include "msCAN.h"

#define MAXMSGLENGTH 8


/* function */
int initCOM(void);

char SendMsg(int id, char *data, unsigned char length);

//int RecMsg(unsigned char msg_id, char * data);
//int RecMsgBlock(unsigned char msg_id,char *data);

unsigned char Received(int *id,char *data, char *lenght); //interrupt handler
function body
```

com.c

```
#include "com.h"

int initCOM(void)
{
  (void)initMSCAN();
  return 0;
}

char SendMsg(int id, char *data, unsigned char length)
{
  if (length < MAXMSGLENGTH) {
    (void)MSCANSendFrame(id,data,length);
    return length;
  }
  else
  return -1;
}

unsigned char Received(int *id,char *data, char *lenght){
  return(MSCANReceived(id,data,lenght));
 }
```

10.2 Knihovna msCAN - výpis programového kódu

mscan.h

```
/*  assembler code macros */
#define SEI asm SEI // disable maskable interrupts
#define CLI asm CLI // enable maskable interrupts

/* MSCAN12 registers */
#define CMCR0 (*((unsigned char*) 0x0100))
#define CMCR1 (*((unsigned char*) 0x0101))
#define CBTR0 (*((unsigned char*) 0x0102))
#define CBTR1 (*((unsigned char*) 0x0103))
#define CRFLG (*((unsigned char*) 0x0104))
#define CRIER (*((unsigned char*) 0x0105))
```

```
#define CTFLG (*((unsigned char*) 0x0106))
#define CTCR (*((unsigned char*) 0x0107))
#define CRXERR (*((unsigned char*) 0x010E))
#define CTXERR (*((unsigned char*) 0x010F))

#define PCTLCAN (*((unsigned char*) 0x013D))
#define DDRCAN (*((unsigned char*) 0x013F))
#define PORTCAN (*((unsigned char*) 0x013E))

/* CMCR0 flags */
#define CSWAI 0x20
#define TLNKEN 0x08
#define SFTRES 0x01

/* CMCR1 flags */
#define LOOPB 0
#define WUPM 0
#define CLKSRC 0

/*  CRFLG flags */
#define WUPIF 0x80
#define RWRNIF 0x40
#define TWRNIF 0x20
#define RERRIF 0x10
#define TERRIF 0x08
#define BOFFIF 0x04
#define OVRIF 0x02
#define RXF 0x01

/* CRIER flags */
#define WUPIE 0x80
#define RWRNIE 0x40
#define TWRNIE 0x20
#define RERRIE 0x10
#define TERRIE 0x08
#define BOFFIE 0x04
#define OVRIE 0x02
#define RXE 0x01

/* PCTLCAN flags*/

#define PUPCAN 0x02 // pull-up mode ?
#define RDPCAN 0x01 // reduced CAN drive mode ?


/*  buffer memory structure */
typedef struct msCAN_rec_buffer{
unsigned int id;
unsigned int dummy;
unsigned char data[8];
unsigned char lenght; // 0..8
 } msCAN_rx_buffer;

typedef struct msCAN_tx_buffer{
unsigned int id;
unsigned int dummy;
unsigned char data[8];
unsigned char lenght;
unsigned char prio;
 } msCAN_tx_buffer;
```

```
/*   message properties*/

#define RTR 0x0010
#define IDE 0x0008

/*****************************************************************
ERROR values
*****************************************************************/
enum ErrorCode {
  NO_ERR,
  ERR,
  NO_MESS,
  ERR_SCHED,
  ERR_RST,
  ERR_MO,
  NO_MO,
  FALSE_ID_LENGHT,
};


/*****************************************************************
Identifier mask registers
*****************************************************************/
#define CIDMR0 (*((unsigned char *) 0x114))
#define CIDMR1 (*((unsigned char *) 0x115))
#define CIDMR2 (*((unsigned char *) 0x116))
#define CIDMR3 (*((unsigned char *) 0x117))
#define CIDMR4 (*((unsigned char *) 0x11C))
#define CIDMR5 (*((unsigned char *) 0x11D))
#define CIDMR6 (*((unsigned char *) 0x11E))
#define CIDMR7 (*((unsigned char *) 0x11F))

/*****************************************************************
Identifier acceptance registers
*****************************************************************/
#define CIDAR0 (*((unsigned char *) 0x110))
#define CIDAR1 (*((unsigned char *) 0x111))
#define CIDAR2 (*((unsigned char *) 0x112))
#define CIDAR3 (*((unsigned char *) 0x113))
#define CIDAR4 (*((unsigned char *) 0x118))
#define CIDAR5 (*((unsigned char *) 0x119))
#define CIDAR6 (*((unsigned char *) 0x11A))
#define CIDAR7 (*((unsigned char *) 0x11B))

/*****************************************************************
Identifier masks registers values
*****************************************************************/
#define ID_AM_0 0x00
#define ID_AM_1 0x00
#define ID_AM_2 0x00
#define ID_AM_3 0x00
#define ID_AM_4 0x00
#define ID_AM_5 0x00
#define ID_AM_6 0x00
#define ID_AM_7 0x00 // ff => budou projmuty vsechny zpravy

/*****************************************************************
Identifier acceptance registers
*****************************************************************/
```

```
#define ID_AC_0 0x00
#define ID_AC_1 0x00
#define ID_AC_2 0x00
#define ID_AC_3 0x00
#define ID_AC_4 0x00
#define ID_AC_5 0x00
#define ID_AC_6 0x00
#define ID_AC_7 0x00

/*********************************************************************
Function prototypes
*********************************************************************/

int initMSCAN(void);
char MSCANSendFrame(unsigned int id, char *data, unsigned char lentgth);
unsigned char  MSCANReceived(int *id,char *data, char *lenght);
```

mscan.c

```
#ifndef _msCAN
#define _msCAN

#include "msCAN.h"

/* defines */
#define PORTH (*((unsigned char *) 0x29))
#define DDRH (*((unsigned char *) 0x2B))

#define NULL                    ((void *) 0)

#define CANOSCFREQ 16000000

/* buffer structures definition */
volatile msCAN_rx_buffer *Rx = (msCAN_rx_buffer *) 0x0140;
volatile msCAN_tx_buffer *Tx0 = (msCAN_tx_buffer *) 0x0150;
volatile msCAN_tx_buffer *Tx1 = (msCAN_tx_buffer *) 0x0160;
volatile msCAN_tx_buffer *Tx2 = (msCAN_tx_buffer *) 0x0170;

/* function definitions */
void _MSCANSetRegEnable(void){
  CMCR0|=SFTRES;
}

void _MSCANSetRegDisable(void){
  CMCR0&=~SFTRES;  // enable control registers setting
}

/*****************************************************************************
Function Name :    _initTiming
Description :  nastaveni casovacich registru sbernice
Return value: void
*****************************************************************************/
void _initTiming(void){
  CBTR0 = 0x49;
  CBTR1 = 0x14;
}

/*****************************************************************************
Function Name :    _MSCANinitTiming
```

```
Description :  nastaveni casovani sbernice
Return value: void
*******************************************************************************/
void _MSCANinitTiming(long int speed, unsigned char sjw,
     unsigned char tseg1, unsigned char tseg2)
     {
       float tq=0,  // time quantum
       bit_time =0; // bit time
       char prescaler = 0; // time prescaler

       bit_time = 1/speed;
       tq =bit_time/(sjw+tseg1+tseg2);
       prescaler = (1/CANOSCFREQ)*tq;

       //CBTR0 = ((sjw-1)<<6) | (prescaler -1);
       //CBTR1 = ((tseg2-1)<<4) | (tseg1-1);

     }
/*******************************************************************************
Function Name :    initMSCAN
Description :  inicializace radice
Return value: 0 - inicializace probehla vporadku
*******************************************************************************/
int initMSCAN (void){
  _MSCANSetRegEnable();
  _initTiming(); // casovani CAN sbernice
  CMCR1 = 0x01; // vnejsi zdroj hodin
  _MSCANSetRegDisable();
  CRIER=0x01;  // maska pro prijem preruseni od Rx bufferu
  CLI;  // globalni povoleni preruseni
  return 0;
}
/*******************************************************************************
Function Name :    _StoreData
Description :  ulozeni dat do prislusneho TX bufferu
Return value: 0 - uspesne odeslana data
*******************************************************************************/
int _StoreDataTx(msCAN_tx_buffer *Tx,unsigned int id, char *data, unsigned char
lenght)
{
  int i=0;
  Tx->id = (id<<5); // standard id
  if (data != NULL ) {  // datovy ramec?
    for (i=0;i<8;i++) Tx->data[i]=0x0;   // vymaz pole dat
    for (i=0;i<lenght;i++)  Tx->data[i]=data[i]; // uloz data do bufferu
    Tx->lenght = lenght; // delka datoveho pole
    return 0;  // uspesny konec
  } else  {   // remote ramec
    Tx->id |=0x0010;  /* RTR = 1*/
    Tx->lenght = 0;
    return 0;
  }
}

/*******************************************************************************
Function Name :    MSCANSendFrame
Description :  odeslani ramce remote i datoveho
Return value: delka odeslanych dat
*******************************************************************************/
char MSCANSendFrame(unsigned int id, char *data, unsigned char length)
```

```
{
  for (;;){    // nekonecna smycka dokud se nepodari odeslat data
    if (CTFLG & 0x01){  // buffer Tx1 prazdny?
      (void)_StoreDataTx(Tx0,id, data, length); // uloz data do bufferu
      CTFLG = 0x01;  // zacatek prenosu dat
      return length; // vrati delku zapsanych dat
    }
    else if(CTFLG & 0x02){ // buffer Tx2 prazdny?
      (void)_StoreDataTx(Tx1,id, data, length);
      CTFLG = 0x02;
      return length;
    }
    else if(CTFLG & 0x04){ // buffer Tx3 prazdny?
      (void)_StoreDataTx(Tx2,id, data, length);
      CTFLG = 0x04;
      return length;
    }
  } // konec for(;;)
}

/*****************************************************************************
Function Name       :   MSCANreceive
Description : Rutina urcena pro nacteni dat z Rx bufferu
Return value: 1..8  data lenght - data frame
= 0  remote frame
*****************************************************************************/
unsigned char  MSCANReceived(int *id,char *data, char *lenght){

  int i=0; // index
  char *tail;

  *id = Rx->id>>5;
  if (Rx->id & 0x02){  // remote frame
    *lenght = 0;
    data =NULL; // zadna data
  }
  else { // data frame
    tail = data;
    for (i=0;i<Rx->lenght;i++)  {
      *tail = Rx->data[i];  // kopie dat z bufferu
      tail++;
    }
    *lenght = Rx->lenght; // delka dat
  }
  CRFLG = 0x01;  // RXF flag -> 0
  return(Rx->lenght);
}

/*****************************************************************************
Function Name       :   MSCANerror
Description : Prerusovaci handler pro obsluhu chybovych stavu radice MSCAN12
*****************************************************************************/

interrupt 27 void MSCANerror(void){
  if (CRFLG & 0x20) { // Transmit warning  96<counter<127
    CRIER &= 0xDF; // zakazani TransmitWarning interrupt
    CRIER |= 0x08; // povoleni TransmitError interrupt
    CRFLG = 0x08; // vymazani TransmitErrorPassive flagu
    CRFLG = 0x04; // reset BussOff flagu
  }
```

```
  if (CRFLG & 0x08) { // Transmit warning  counter >127
    CRIER &= 0xF7;
    CRIER |= 0x24;
    CRFLG = 0x20;
    CRFLG = 0x04;
  }
  if (CRFLG & 0x04) { // Buss off Transmit  counter  >255
    CRIER &= 0xFB;
    CRFLG = 0x04;
    CRIER |= 0x08;
  }

  if (CRFLG & 0x40) { // Receive warning     96<counter<127
    CRIER &= 0xBF;
    CRIER |= 0x10;
    CRFLG = 0x10;
  }

  if (CRFLG & 0x10) { // Buss off Transmit  counter  >255
CRIER &= 0xEF;
CRIER |= 0x40;
CRFLG = 0x40; // reset receive warning flag
  }
}

#endif
```

10.3 Knihovna adc - výpis programového kódu

adc.h

```
#ifndef _adc_
#define _adc_

/* A/D CONTROL REGISTERS */

#define ATDCTL2 (*((unsigned char *) 0x62))
#define ATDCTL3 (*((unsigned char *) 0x63))
#define ATDCTL4 (*((unsigned char *) 0x64))
#define ATDCTL5 (*((unsigned char *) 0x65))
#define ATDSTAT1 (*((unsigned char *) 0x66))
#define ATDSTAT2 (*((unsigned char *) 0x67))
#define ATDSTH (*((unsigned char *) 0x68))
#define ATDSTL (*((unsigned char *) 0x69))
#define ADRX0H (*((unsigned char *) 0x70))
#define ADRX0L (*((unsigned char *) 0x71))

/* PORT REGISTERS*/
#define PORTA (*((unsigned char *)0x00))
#define DDRA (*((unsigned char *)0x02))
#define PUCR (*((unsigned char *)0x0C))

/* INIT VALUES */
#define ATDCTL2_VALUE 0xC0
#define ATDCTL3_VALUE 0x03
#define ATDCTL4_VALUE 0x01
#define ATDCTL5_VALUE_1 0x00
#define ATDCTL5_VALUE_2 0x01

/* FUNCTION PROTOTYPES */
```

```
/* ADC initialization*/
int InitADC(void);

/* return ADC conversion result*/
unsigned char StartADC(void);

#endif
```

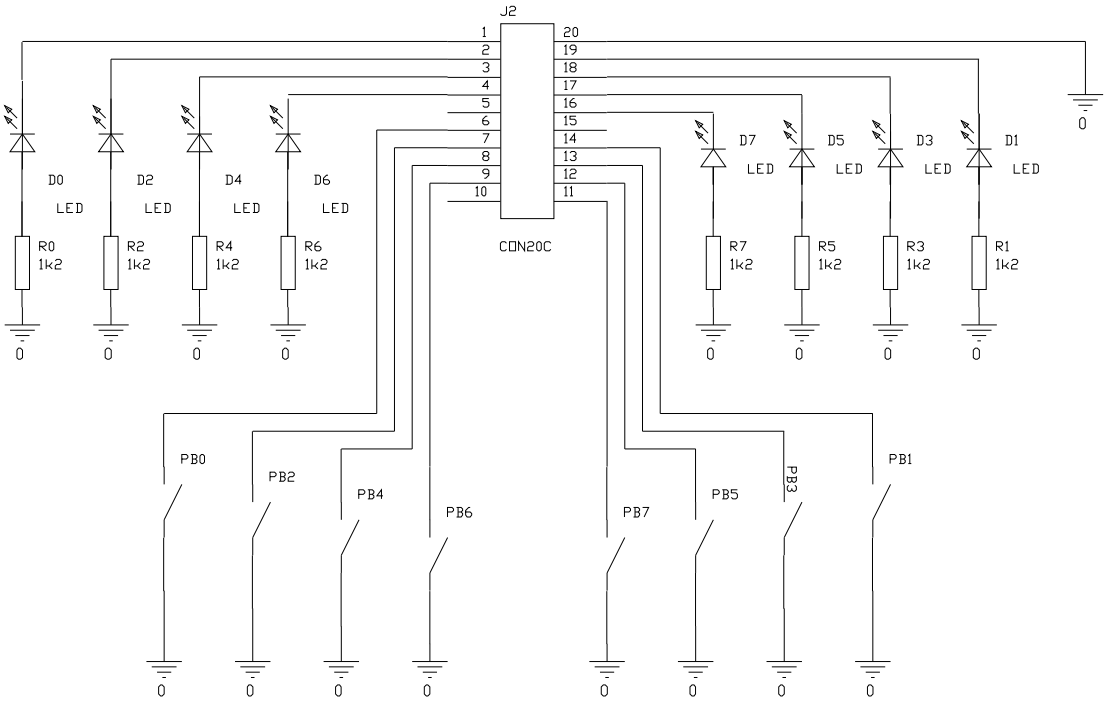
adc.c

```
#include "adc.h"

/* ADC initialization*/
int InitADC (void)
{
  ATDCTL2 = ATDCTL2_VALUE;
  ATDCTL3 = ATDCTL3_VALUE;
  ATDCTL4 = ATDCTL4_VALUE;
  return 0;
}

/* ADC conversion function */
unsigned char StartADC(void)
{
  unsigned char result =0;
  ATDCTL5 = ATDCTL5_VALUE_2;
  while ((ATDSTAT1&0X80)==0);
  result = ATDSTH;
  return result;
}
```

11 Příloha B

11.1 Karta DI/DO - schéma zapojení

Karta DI/DO slouží k vizualizaci stavu portu a zároveň jako karta digitálních vstupů. Pomocí tlačítek je možno ovlivňovat chod řídícího procesu CAN modulu. Schéma zapojení je uvedeno na následujícím obrázku.



12 Příloha C

12.1 Příklad - použití komunikačních funkcí

V této části dokumentu je uveden příklad použití komunikačních primitiv implementovaných v knihovně `com.h`, kterou jsem naprogramoval v rámci této diplomové práce. Karta `Sender` v případě stisku jednoho z tlačítek připojených k portu G zobrazí aktuální stav na port H a odešle prostřednictvím sběrnice CAN zprávu s identifikátorem `MSG_ID` a s daty obsahujícími aktuální stav portu digitálních vstupů. Karta `Receiver` přijme data a zobrazí jejich obsah pomocí modulu digitálních výstupů na port H.

Aplikace Sender

Při implementaci aplikačního kódu jsem postupoval následovně:

1. podle postupu pospsaného v kapitole 5 v aplikaci CodeWarrior založíme aplikaci Sender, nastavíme linker, tak aby mezi spustitelnými soubory byly `*.sx` a nahradíme automaticky vygenerovaný soubor `default.prm`

2. dvojitým poklepáním myší na soubor `osdef.oil` spustíme konfigurační nástroj OSEKbuilder v objektu OS nastavíme atribut `TargetMCU` na hodnotu `HC12D60` a přidáme událost(event) `SendStateEvent` a jako jejího majitele novou úlohu `SenderTask`. Postup a nastavení parametrů systémových objektů byl popsán v kapitole 3 a proto se jím na tomto místě nebudeme dále zabývat. Tímto postupem jsem tedy deklarovali rozšířenou úlohu, která je vlastníkem události `SendStateEvent`. Uložíme soubor a přejdeme zpět do programu CodeWarrior.

3. do projektu vložíme soubory `com.c` a `msCAN.C`, které umožňjí inicializaci komunikační vrstvy,

4. Následující změny jsou prováděny souboru `main.c`,

5. namapujeme a vyřadíme z činnosti obvod watch dogu pomocí registru COPCTL,

6. namapujeme port H a G a pomocí odpovídajících DDR registrů je nastavíme tak, aby port H byl výstupní a port h byl vtstupní,

7. v těle úlohy `InitTask` použijeme funkci `initCOM`, která zabezpečí inicializaci řadiče msCAN,

8. v deklarační části použijeme systémové služny `DeclareTask` a`DeclareEvent` pro deklaraci systémových objektů `SendStateEvent` a `SendTask`,

9. nadefinujeme tělo úlohy `SendTask`. Úloha bude obsahovat nekonečnou smyčku, ve které bude čekat na ativaci události `SendStateEvent`. Doje-li k ativaci vyšle pomocí primitivy `SendMsg` zprávu s iodentifikátorem `MSG_ID` a kopií vstupního portu H.

10. v případě, že dojde ke stisku tlačítka na portu vstupů a tím ke změně portu G dojde v těle úlohy `CyclicTask` k aktivaci události `SendStateEvent` a tedy i ke zpuštění úlohy `SendTask` a k odeslání zprávy.

11. použitím funkce build nebo funkční klávesy `F7` programu CodeWarrior přeložíme aplikační kód. Výsledný spustitelný soubor je umístěn v adrešáři `bin` s koncovkou `*.sx`.

12. Pro nahrání a spuštění aplikačního kódu do FLASH paměti CAN modulu, použijeme program VLoader podle postupu posaného v [7]. Nepodaří-li se navázat spojení s přípravkem CAN-modulů, doporučuji zkontrolovat nastavení přenosové rychlosti, portu a stav propojovacího konektoru `JP6` (viz dále).

13. Po stistku tlačítka je opdeslána zpráva (LED-diody vysílače blikají).

14. Chceme-li, aby po zapnutí zařízení došlo k automatickému spuštění řídící aplikace, použijeme propojovací konektor (jumper) `JP6`. Je-li propojen, dojde po přivedení napaájecího napětí k modulu ke spuštění aplikace.

Aplikace Receiver

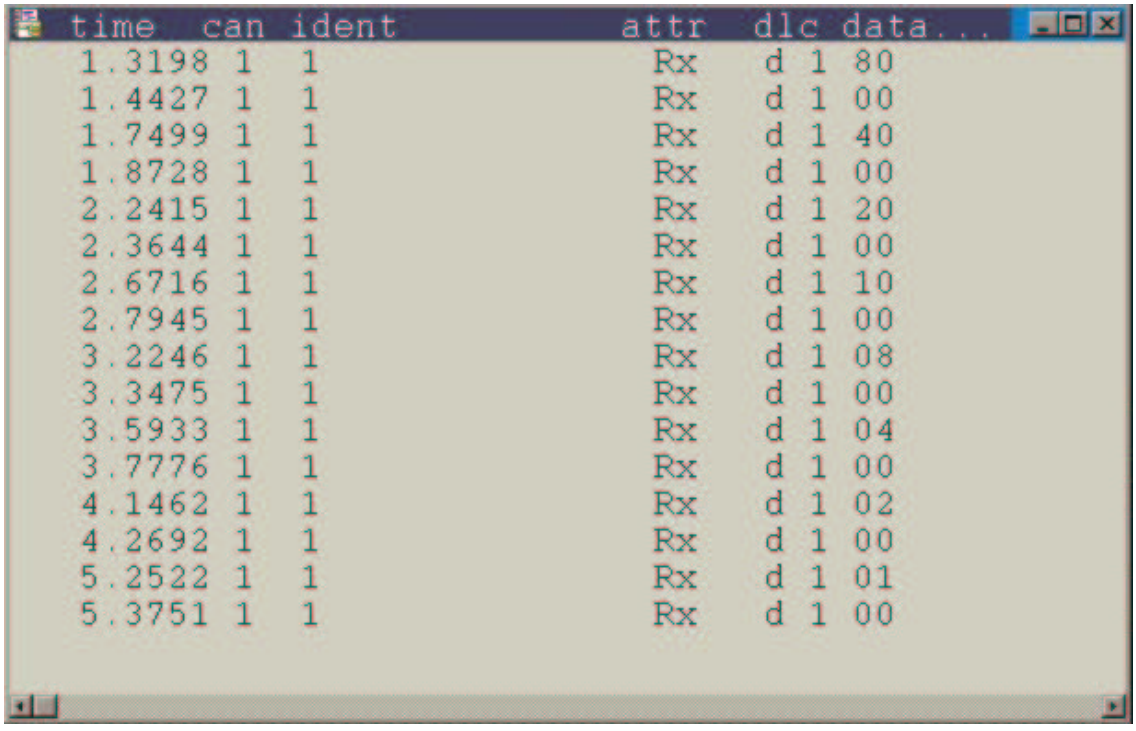
Tato aplikace zajistí příjem zprávy a zobrazení obsahu dat zprávy na port H.

1. postup založení aplikace je shodný s postupem posaným výše,

2. namapujeme a vyřadíme obvod watch-dogu, namapujeme port H a nastavíme jej jako výstupní, změníme obsahu souboru `default.prm` a přidáme těla knihoven `msCAN.c` a `com.c` do projektu.

3. podle postupu posaného v kapitole 5 přidáme do aplikace novou obslužnou rutinu přerušení `MsCANISR`. V těle této rutiny použijeme pro příjem dat funkci `Received` z knihovny com a přijatá data zobrazíme na port H.

4. aplikační kód známým způsobem zkompilujeme a nahrajeme do FLASH paměti CAN modulu.



Obrázek 24: Příklad - výpis z CAN analyzátoru

V této fázi máme obě zařízení připravená pro komunikaci. V okamžiku stisku tlačítka digitálních vstupů karty `Sender` dojde k zobrazení stavu portu digitálních vstupů na port G a k odeslání zprávy s identifikátorem `MSG_ID` prostřednictvím sběrnice CAN do modulu `Receiver`, kde je zpráva přijmuta a zobrazena na portu. Na obrázku 24 je výpis z CAN analyzátoru, který jsem připojil na sběrnici CAN. Vidíme zde zprávy s identifikátorem 0x01 (`MSG_ID`) a s jedním bytem kopie stavu portu G.

Výpis sprogramového kódu obou aplikací je uveden na přiloženém CD v adresáři `/example2`. Pro případné zájemce o kód uvedený na CD doporučuji zkopírovat obsah libovolného adresáře na pevný disk. Aby bylo možno program zkompilovat v programu CodeWarrior doporučuji změnit cesty v souboru `options.h` umístěném v hlavním adresáři zdrojových kódů `Sender` nebo `Receiver`.

Obsah přiloženého CD

Nedílnou součástí této diplomové práce je CD, které obsahuje následující adresáře:

- `CCU`
  - `CAR` - zdrojové kódy řídící aplikace karty CAR,
  - `CUAC` - zdrojové kódy řídící aplikace karty CUAC,
  - `TU` - zdrojové kódy řídící aplikace karty TU,
  - `shared` - knihovny,
- `dokumentace`
  - `CAN` - specifikace CAN
  - `Metroerks` - materiály k turboOSEK a COdeWarrior,
  - `Motorola` - dokumentace k mikropočítači HC12D60,
  - `Osek` - specifikace OSEK/VDX,
  - `Philips` = dokumentace k budiči fyzické vrstvy CAN Hi speed PCA82C250,
- `dp_tex` - zdrojové kódy tohoto dokumentu v systému LaTeX 2_ε,
- `priklady`
  - `priklad 1` - zdrojové kódy obsahující názorný příklad použití modulu DI/DO
  - `priklad 2` - zdrojové kódy k příkladu Sender-Receiver,