CAM Programmeren voor dummies

"Licht in de duisternis van programmatuurontwikkeling voor CAMS en smartcards"

Versie 5.0

Inhoudsopgave

1. Inleiding

Ontvangst via een schotel en receiver was vier maanden geleden nog geheel nieuw voor mij. De eerste maanden ben ik bezig geweest met mijn decoder, een Manhattan Mx met geïntegreerde Matrix Reloaded CAM. Geweldig op gang geholpen door mijn leverancier kon ik al snel wat spelen met verschillende cams, emulaties, titanium en fun kaarten.

Al snel raakte ik geïnteresseerd in de interne werking van de ontvanger, cams en smartcards. De informatie hierover is schaars en wordt wellicht om commerciële redenen ook wel bewust schaars gehouden. Ik ben daarom begonnen met het verzamelen van documentatie en die te bestuderen. Het uiteindelijke doel is volledig inzicht te krijgen in de werking van satellietuitzendingen, de ontvanger, de cams en de smartcards. Gezien mijn persoonlijke achtergrond betreft dit met name het softwarematige aspect hiervan, en dan in het bijzonder het ontwikkelen van firmware voor cams en smartcards. In dit document zullen we dan ook toewerken naar zelf ontwikkelde firmware, de broncode hiervan en een toelichting hierop.

Gaandeweg kwam ik er ook achter dat er juridische regelgeving is die dit onderzoek tot een bepaalde diepgang beperkt. Vandaar dat de doelstelling bij versie 5 van dit document ook is bijgesteld van een volledige emulatie tot werkende firmware zonder emulatie.

Hoewel de doelstelling software ontwikkeling is, bevat het document bijzonder veel informatie die ook voor niet-programmeurs interessant is. Vooral de eerste hoofdstukken bevatten veel achtergrond informatie. Er zijn veel verwijzingen opgenomen naar documentatie, indien mogelijk met een directe internet pagina. Hierdoor is het mogelijk de documentatie te bestuderen op grond waarvan dit document is samengesteld.

Gedurende dit proces heb ik veel mensen lastig gevallen met vragen. Van veel mensen heb ik ook bijzonder veel steun en informatie mogen ontvangen. Ik wil dan ook iedereen die heeft meegeholpen van harte bedanken! Mijn avontuur in satellietland is nog lang niet beëindigd. Of er ooit een definitieve versie van dit document zal komen, kan ik dan ook niet beloven!

Hermanator
14 maart 2005

Met dank aan: Duwgati, Pic-o-matic, MiLo, Ozzo, EnEmA, John43, Avensis, mr ed, Cammy en anderen.

Speciale dank aan: Bommeltje, Wildcard, Ricow, Zilverster en Superpippo82xxx.

2. Een globaal overzicht

2.1 MPEG-2

Een uitzending waarnaar je via de schotel kijkt is eigenlijk een digitaal **MPEG-2** bestand zoals ook de talloze filmpjes waarmee we elkaar bestoken per e-mail. Het bestand wordt in dat geval uit het mailtje gehaald en opgeslagen op je harddisk, waarna het met bijvoorbeeld de Mediaplayer van Windows wordt afgespeeld. Een gesimplificeerde voorstelling van uitzendingen per satelliet is dat dit bestand nu als *datastream* wordt verzonden vanaf de satelliet. Als het ware alsof je het bestand download van internet, maar nu via de schotel. De satellietontvanger speelt deze datastream (het bestand dus) direct tijdens het ontvangen al af op de televisie. Deze datastream kun je zien als een lange reeks bytes van het oorspronkelijke MPEG-2 bestand, netjes gegroepeerd in **packets,** met hierin toegevoegde controle-informatie om onder andere een correcte ontvangst te garanderen (error correctie, etc). Een vergelijkbare werking als bij streaming video over internet dus eigenlijk.

De werkelijkheid is vanzelfsprekend iets complexer. Een satelliet bestaat bijvoorbeeld uit verschillende transponders die op verschillende frequenties meerdere **Transportstreams** kunnen uitzenden met daarbinnen meerdere substreams (**PES, Packetized Elementary Stream**, geïdentificeerd met een **PID nummer**. Packets met hetzelfde PID nummer behoren dus tot dezelfde PES. Verschillende PES met ieder hun eigen soort inhoud, vormen tezamen het programma waar we naar kijken (beeld, geluid, EPG, etc.). Deze programma's kunnen weer over meerdere satellieten *logisch* zijn gegroepeerd in een boeket, waarop men zich kan abonneren. In de transportstream bevinden zich tabellen, waarin de ontvanger o.a. kan vinden uit welke PES (PID nummers) een programma bestaat en in welke PES de voor het ontsleutelen van het gekozen programma benodigde gegevens te vinden zijn. Ook gegevens voor het bijwerken van de abonnementsgegevens van de abonnees worden op deze manier in een PES verzonden. De ontvanger filtert al deze berichten uit de datastream om de programma's correct op de televisie te kunnen tonen en om de abonnementsgegevens op de smartcard bij te werken. In appendix 1 zijn meer details over de transportstream te vinden.

2.2 Scrambling

Het uitzenden via de satelliet is in Europa gestandaardiseerd in de **DVB-s** specificaties (**Digital Video Broadcast via Satelliet**). Zo bestaan er bijvoorbeeld ook standaards voor uitzending via de kabel, **DVB-c** en via de ether, **DVB-t** (terrestrial). De digitalisering van het beeldmateriaal vindt plaats volgens de MPEG-2 standaard. Om storingen van het satelliet signaal te voorkomen én om ervoor te kunnen zorgen dat alleen bevoegden de uitzending kunnen zien, wordt de datastream **scrambled** (gecodeerd) verzonden. De scrambling is gestandaardiseerd in het DVB Common Scrambling Algorithm (**CSA**).

De CSA scrambling voorziet in een gestandaardiseerde versleuteling (scrambling) van de inhoud van de packets in een PES met zogenaamde **controlwords**. Aan de hand van een controlword wordt bij de verzending de CSA algoritmische bewerking op de packets uitgevoerd, waardoor de inhoud een onherkenbare reeks tekens wordt. In de satellietontvanger kunnen, mits de voor de scrambling gebruikte controlwords bekend zijn, via de omgekeerde bewerking de packets weer worden teruggezet naar de oorspronkelijke herkenbare MPEG-2 informatie. De ontvanger kan deze dan op de televisie afspelen. Iedere 2 tot 10 seconden wordt bij het uitzenden een nieuw, *willekeurig* controlword gegenereerd door een random controlword generator. Dit controlword wordt dan weer gebruikt voor de scramble van de volgende 2 tot 10 seconden, waarna weer een nieuw random controlword wordt gegenereerd.

Het CSA algoritme is dus universeel voor alle DVB uitzendingen en is geïmplementeerd in alle hiervoor gemaakte ontvangers. Ondanks dat alle ontvangers dus met hetzelfde algoritme werken, zijn ze echter niet in staat ons naar een programma te laten kijken als ze niet over de juiste controlwords beschikken. Alleen door het omgekeerde algoritme toe te passen, met de juiste controlwords op de juiste delen van de PES stream, hebben we beeld.

2.3 ECM Entitlement Control Message

Het controlword wordt dus om de 2 tot 10 seconden vervangen door een random controlword generator bij het verzenden. **Alleen met het juiste controlword, op het juiste moment, hebben we dus gedurende 2 tot 10 seconden beeld.** Om ons ongestoord naar een programma van bijvoorbeeld een uur te kunnen laten kijken, moet de ontvanger dus over zo'n 360 tot misschien wel 1800 verschillende controlwords beschikken! Maar deze controlwords worden op min of meer hetzelfde tijdstip op een geheel andere locatie dan bij ons thuis, willekeurig gegenereerd met een frequentie van 2 tot 10 seconden. Hoe moeten deze controlwords dan in onze ontvanger komen? Het antwoord is simpel. De controlwords worden in een aparte PES, dus met een apart PID nummer, in een speciaal soort berichten (packets) meegezonden: de **ECM's** (**Entitlement Control Message**). Deze ECM's worden uit de datastream gefilterd door de ontvanger. Het controlword wordt uit het ECM bericht gehaald en hiermee kan weer 2 tot 10 seconden PES packets worden descrambled en op de televisie worden getoond. Tot natuurlijk het volgende ECM bericht wordt ontvangen met weer een nieuw controlword. Overigens wordt ook andere, op dit moment voor ons begrip nog niet relevante informatie over de uitzending, met ECM berichten verzonden.

2.4 Encryptie / Keys

De controlwords worden dus om de 2 tot 10 seconden meegezonden in de PES zodat onze ontvanger op ieder moment de beschikking heeft over het actuele controlword. Het verstrekken van deze controlwords is de plaats waarop men de bevoegdheid tot het kunnen kijken naar bepaalde programma's heeft geregeld. Bij een uitzending voor een beperkt publiek, de abonnees, wordt het controlword namelijk versleuteld met een apart algoritme. Deze versleuteling van controlwords noemen we **encryptie** en daar zijn meerdere soorten van ontwikkeld, zoals Seca, Conax en Irdeto. De provider, de uitzender van het programma, heeft gekozen voor één of meerdere van deze encrypties om de ecm's mee te versleutelen. Hierdoor zijn alleen de ontvangers die in staat zijn de controlwords volgens deze encryptie te ontsleutelen, in staat om het programma op de televisie te tonen.

Er is dus een duidelijk verschil tussen scrambling en encryption:

- **Scrambling** is de standaard DVB-s CSA codering, waarbij alle uitzendingen worden versleuteld met controlwords.
- **Encryption** is een apart algoritme om de meegezonden controlwords te beveiligen, zodat de uitzending alleen descrambled kan worden door hiertoe gerechtigde kijkers.

Bij een **FTA (Free To Air)** uitzending die iedereen dus vrij mag ontvangen, worden de controlwords unencrypted in ECM's verzonden. Iedere DVB ontvanger is dus in staat om de controlwords uit de ECM's te halen en het betreffende programma te descrambelen. Ook een FTA ontvanger heeft dus wel degelijk een descrambler. Alleen is een FTA ontvanger niet in staat om een encrypted ECM te ontsleutelen.

Als een provider uitzendt in meerdere coderingen (zoals momenteel bijvoorbeeld bij Nederlandse zenders Seca en Irdeto), dan worden gewoon voor beide coderingen ECM berichten verzonden met de juiste controlwords. Een seca ontvanger decrypt dan bijvoorbeeld de seca ECM's en negeert de Irdeto ECM's, terwijl een Irdeto ontvanger de Irdeto ECM's decrypt en de seca ECM's negeert. Beiden verkrijgen echter op deze manier dezelfde controlwords waarmee het programma kan worden descrambled.

Om de encrypted controlwords uit de ecm's te kunnen ontsleutelen zijn keys (sleuteltjes) benodigd. Dit zijn de keys waar we altijd zo druk mee in de weer zijn. Met de juiste keys kan de ontvanger dus de controlwords uit de ecm's decrypten en met die controlwords weer het programma descramblen zodat wij het kunnen zien. Deze sleutels worden, afhankelijk van de provider, periodiek veranderd. Bij sommigen is dat één keer per maand, terwijl we bijvoorbeeld bij conax een hele tijd iedere week op dinsdag een wisseling hebben gehad.

Helaas voor de hackers zijn de sleuteltjes niet de enige factoren van een encryptie die de provider flexibel kan variëren. De encryptie vindt plaats met de keys maar ook met gebruikmaking van bijvoorbeeld hash tabellen, die dan ook bekend moeten zijn om het volledige algoritme uit te kunnen voeren.

Bij keys hebben we hetzelfde probleem als bij de controlwords. Hoe komen deze nu in de ontvanger? Hoewel de wijzigingsfrequentie veel lager is dan bij de controlwords, moeten wel enorm veel abonnees bijvoorbeeld wekelijks van nieuwe keys worden voorzien. Je raad het al, ook deze worden in de datastream in een PES (op een PID nummer dus) meegezonden in een apart soort bericht: de EMM.

2.5 EMM (Entitlement Management Message)

Hoe komen deze periodiek wijzigende keys dan weer in de ontvangers van de abonnees? Hiervoor is een nieuwe soort berichten bedacht: de **EMM (Entitlement Management Message)** berichten. Ook deze berichten worden met dezelfde encryptie (seca, irdeto, etc.) versleuteld en op een PID nummer verzonden. Maar aangezien de EMM bedoeld is om alleen diegenen met een geldig abonnement te voorzien van o.a. nieuwe keys, worden deze met een per abonnement of abonnementsgroep unieke masterkey versleuteld.

Een abonnee heeft een **smartcard** met abonnementsgegevens, waaronder deze **masterkey**. Een EMM bericht met onder andere de nieuwe keys wordt nu encrypted met deze masterkey. Een EMM bericht kan dus uitsluitend met behulp van deze ene specifieke masterkey worden ontsleuteld. Zo is men dus in staat om één enkel(e groep) abonnement(en) te activeren of te deactiveren. De nieuwe keys in het EMM bericht worden bijgewerkt op de smartcard. Met de EMM's worden nog andere kenmerken van het abonnement en het algoritme verzonden en bijgewerkt op de smartcard. De codering kent daarom naast het algoritme ook commando's om de kaart te bewerken. Samenvattend:

1. Een **PES,** geïdentificeerd met een **PID**nr wordt ge**scrambled** met **controlwords;**
2. Die controlwords worden op een aparte PID meegezonden in **ECM berichten;**
3. Deze ECM berichten worden bij "beschermde"uitzendingen **encrypted** met **keys;**
4. Deze keys worden periodiek verzonden in een **EMM bericht** per abonnement(sgroep);
5. Een EMM bericht wordt encrypted met een per abonnement(sgroep) unieke **masterkey;**
6. Deze masterkey staat op de **smartcard** en hierop worden de keys bijgewerkt.

De smartcard bevat dus de gegevens van het abonnement en de voor de decryption van de controlwords benodigde keys, algoritme, hashtables, etc. Globaal is de werking hiervan (bij een ontvanger zonder CAM) als volgt. Als wij in de ontvanger kiezen voor een programma dat encrypted wordt uitgezonden, dan zal de ontvanger continu de ecm's hiervoor uit de juiste PES filteren. Vervolgens vraagt de ontvanger de smartcard het controlword uit een ecm te decrypten. De smartcard decrypt het controlword en geeft deze terug aan de ontvanger, die hiermee de descramble van (de eerstvolgende 2 tot 10 seconden van) het programma kan doen en dit op ons televisiescherm kan afspelen. Beschikt de smartcard dus niet over de juiste gegevens, dan kan hij de controlwords niet decrypten en blijft ons scherm zwart. Wordt een EMM ontvangen met nieuwe gegevens van een abonnement, dan wordt dit bericht doorgegeven aan de smartcard. In de EMM bevinden zich instructies voor wijziging van de smartcard, de zogenaamde NANO's waardoor de smartcard de juiste actie kan ondernemen. Is de smartcard in staat om de EMM met de masterkey te decrypten, dan kunnen de commando's worden uitgevoerd en worden de gegevens in de smartcard bijgewerkt.

2.6 Cam (Conditional Access Module)

Wat is nu de plaats van de CAM in dit hele proces? Bij oudere ontvangers zit alle genoemde functionaliteit in de ontvanger. Deze kent dan ook meestal maar één encryptie zoals bijvoorbeeld de door Canal Digitaal "goedgekeurde" ontvangers. Het enige dat dit type ontvanger nog nodig heeft is een geldige smartcard. Met dit type ontvanger kunnen we dan ook uitsluitend FTA uitzendingen bekijken en de uitzendingen die zijn gecodeerd in de encryptie die de ontvanger kent.

Een uitgebreidere ontvanger heeft één of meerdere zogenaamde **CI slots (Common Interface)**. Een CI slot is een sleuf in de ontvanger waarin een zogenaamde **CAM (Conditional Access Module)** kan worden gestoken. Een CAM heeft de vorm van een PCM/CIA module met een sleuf waarin een smartcard kan worden geschoven. De benodigde functionaliteit om de encryptie aan te kunnen zit nu in de CAM. Zo zou je bijvoorbeeld een Conax Cam of een Seca Cam kunnen aanschaffen. Hierdoor hoef je voor iedere encryptie in ieder geval geen aparte ontvanger meer te hebben. Je verwisselt gewoon de CAM. Het enige dat je dan nog nodig hebt is een smartcard met een abonnement voor de programma's die je wilt kunnen zien.

Om er onder andere voor te zorgen dat fabrikanten van ontvangers en fabrikanten van CAM's onafhankelijk van elkaar producten kunnen ontwikkelen die toch met elkaar samenwerken, is het uitwisselingsprotocol (de interface) tussen de ontvanger en de CAM gestandaardiseerd. Dit zijn de specificaties voor het CI slot en deze zijn vastgelegd in "*EN 50221 Common Interface Specification for Conditional Access and other Digital Video Broadcasting Decoder Applications*". Omdat dit de specificaties zijn van hoe de CAM communiceert met de ontvanger, zijn dit voor onze studie ongelooflijk belangrijke specificaties.

2.7 UCAS CAM

Nog fraaier is de **UCAS (Universal Common Access System) CAM** die meerdere coderingen aankan**,** zoals bijvoorbeeld de Magic Module, de Matrix, de Xcam en de Dragon. Nu hebben we in principe nog maar één CAM nodig in onze ontvanger. En om het "ongemak" van geldige abonnementskaarten te voorkomen, hebben deze CAM's vaak software aan boord, die niet eens een smartcard nodig hebben. Deze software noemt men een **emulatie** (net doen alsof ze een smartcard aan boord hebben). Het is deze emulatie die we downloaden van internet en in de CAM laden met bijvoorbeeld onze CAS interface. De sleuteltjes die normaal van de kaart worden gehaald, zijn nu in het geheugen van de CAM geladen in de emulatie. Deze sleuteltjes kunnen in de meeste emulaties ook met de afstandsbediening van de ontvanger worden bewerkt. En als de software in de CAM er zelf niet uitkomt, kan hij altijd nog te rade gaan bij een titaniumkaart, funcard of andere all purpose card met gegevens van diverse providers. Het kan zijn dat sommige encrypties zelfs auto update (AU) zijn. De emulatie is dan zo goed geprogrammeerd dat hij de keys van deze encrypties zelf uit de EMM's in de datastream kan filteren, decrypten en kan bijwerken in het geheugen. Bij de recente encrypties is dat echter al enige tijd niet meer het geval.

2.8 En nu verder…

In dit hoofdstuk hebben we de algemene werking van uitzendingen via de satelliet in kaart gebracht en de plaats van de CAM hierin bepaald. In het volgende hoofdstuk gaan we de interne werking van de CAM en de wijze waarop de CAM met de ontvanger communiceert, nader onderzoeken.

Literatuurverwijzingen:

http://www.duwgati.com
http://www.duwgati.com/archive/Documentation/CAS-model.pdf
http://users.pandora.be/satelliet/mpegtrans.pdf
http://www.videoaudioreport.nl/index.php?action=1&catno=57&artno=1079&category=2004-april

3. De CAM: het model

3.1 Inleiding

In het voorgaande hoofdstuk hebben we een globaal inzicht gekregen in de wijze waarop door de providers de digitale uitzendingen via de satelliet worden gescrambled, encrypted en verzonden. Tevens hebben we gezien wat de plaats en de functie van de CAM hierin is. Nu is de tijd gekomen om ons vergrootglas op de CAM te richten en de onderdelen en de werking van de CAM nauwkeuriger in kaart te brengen.

Zoals we in het voorgaande hoofdstuk hebben gezien is de CI interface, het slot waar we de CAM in doen, gestandaardiseerd. Het document met deze beschrijving, "*EN 50221 Common Interface Specification for Conditional Access and other Digital Video Broadcasting Decoder Applications*" is dan ook het uitgangspunt voor dit onderdeel van onze studie. Het overnemen van alle details uit EN50221 zou niet alleen overbodig zijn, maar zou ook niet bijdragen aan de leesbaarheid van dit rapport. Aan bestudering van EN50221 naast dit rapport, kan dus helaas niet worden ontkomen! Hopelijk zal dit rapport de bestudering van EN50221 wel vergemakkelijken.

3.2 Wat doet een cam nu precies?

Een CAM heeft twee logische interfaces: de **Transportstream Interface** en de **Command Interface**. Na een correcte initialisatie van de CAM in de ontvanger, laat de ontvanger de *volledige* transportstream door de CAM lopen over de **Transportstream Interface**. De gehele transportstream stroomt dus door de Transportstream Interface de CAM in en verlaat deze ook weer via de transportstream interface. De functie van de CAM is het bewerken van de transportstream interface in opdracht van de ontvanger. De opdrachten tot bewerking van de datastream worden door de ontvanger aan de CAM verstrekt over de **Command Interface**. Beide logische interfaces lopen over *dezelfde* fysieke interface: de **PCMCIA Interface**. De werking van zowel de Transportstream Interface, de Command Interface als de PCMCIA interface is gedefinieerd in de EN50221 specificaties. Vanuit de EN50221 specificaties wordt wel verwezen naar toepassing van andere standaarden zoals bijvoorbeeld de ISO13818-1 specificaties van de samenstelling van de Transportstream als de specificaties van bijvoorbeeld de PC-Card interface. EN50221 is dus een complete definitie van de CI interface(s) in de ontvanger.

- **Transportstream Interface**: hierover stroomt de gehele transportstream de CAM in en uit
- **Command Interface**: hierover communiceren de ontvanger en de CAM met elkaar
- **PCMCIA interface**: de fysieke interface (connector) voor zowel Transportstream interface als Command Interface

Terwijl de transportstream over de Transportstream Interface door de CAM stroomt, communiceren de CAM en de ontvanger met elkaar over de Command Interface over de uit te voeren bewerkingen op de transportstream.

Wat zijn dan deze bewerkingen die de CAM op verzoek van de ontvanger op de transportstream verricht? Dit betreft het descramblen van die delen van de transportstream, die de ontvanger nodig heeft om het/de door ons gekozen programma(s) te tonen. Dit zijn dus de programma's waarvan de ECM's zijn versleuteld met een encryptie die de ontvanger zelf niet kent. In geval van bijvoorbeeld een **FTA (Free to Air)** uitzending zal de ontvanger de CAM dan ook geen verzoek tot descrambling doen. De ontvanger kan de CAM echter wel om gelijktijdige descrambling van meerdere programma's verzoeken. Voor de CAM is dat niet boeiend, hij bewerkt dan gewoon een groter deel van de transportstream. De CAM is dus eigenlijk gewoon een dataprocessor.

In tegenstelling dus tot wat er soms wordt beweerd, vindt er wel degelijk descrambling plaats in de CAM!

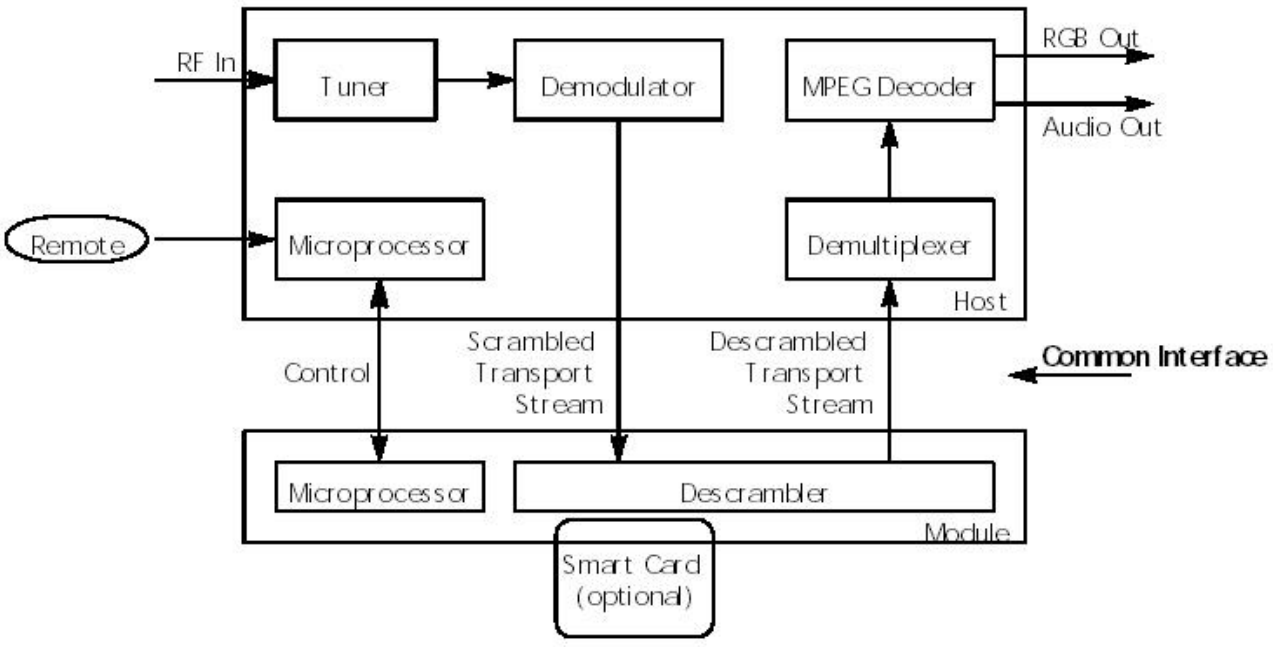
De CAM vertelt de ontvanger tijdens de initialisatie via de Command Interface welke coderingen hij kent. Indien de ontvanger beschikt over meerdere CI slots, dan laat de ontvanger de transportstream als een ketting door iedere CAM stromen. De ontvanger bepaalt vervolgens welke CAM hij via de betreffende Command Interface om de descramble van een bepaald programma verzoekt.

De PES en de packets hiervan hebben in de header een veld “scrambling control flag”. Bij de door de CAM descrambled PES wordt dit veld op ‘00’ gezet. Hierdoor weet de ontvanger dat deze packets direct kunnen worden “afgespeeld”. De andere packets worden door de CAM ongemoeid gelaten, en verlaten de CAM via de Transport Interface weer ongewijzigd. De CAM bewerkt dus delen van de transportstream.

Om de PES te descramblen moet de CAM dus in staat zijn om de hiervoor benodigde ECM’s uit de transportstream te filteren om hieruit de encrypted controlwords te halen en te decrypten. De hiervoor benodigde gegevens zoals de PID waarop de ECM’s kunnen worden gevonden en de toegepaste encrypte, worden oor de ontvanger over de Command Interface aan de CAM verstrekt. Verder heeft de CAM dus de eerder vermelde keys en het algoritme nodig. In een normale UCAS CAM zonder emulatie zal de CAM, net als een ontvanger zonder CI slot, de smartcard verzoeken de controlwords te decrypten. Met deze decrypted controlwords verzorgt de CAM vervolgens de descrambling van de PES. In een CAM met een emulatie heeft de CAM zelf deze gegevens in zijn geheugen, en kan dus zonder smartcard de descramble van de PES verzorgen. Als de firmware van de CAM goed is geprogrammeerd, zullen we een combinatie van beide methoden aantreffen. Het is dan bijvoorbeeld mogelijk per codering in te stellen of er met een smartcard moet worden gewerkt of dat de emulatie gebruikt moet worden.

Samenvattend: de functie van de CAM is dus op aanwijzing van de ontvanger bepaalde delen van de transportstream te descramblen zodat de ontvanger deze kan afspelen. De CAM filtert hiertoe de benodigde ECM’s uit de transportstream, decrypt de hierin aanwezige controlwords en verzorgt de descramble van dat deel van de transportstream waarom de ontvanger heeft verzocht: de gecodeerde programma(‘s) waar wij naar willen kijken.

Onderstaand schema geeft globaal de structuur weer van de CAM en de ontvanger en de uitwisseling tussen beiden via de CI-interface.



3.3 De structuur van de CAM

Nu we globale werking van de CAM in de vorige paragraaf hebben bekeken wordt het tijd eens de onderdelen van de CAM in beeld te brengen.

Links zien we de **host** (receiver) en rechts de **module** (CAM) die met elkaar communiceren over de CI interface in het midden (**PC card interface**)

Boven zien we dat de MPEG-2 **transportstream** de CAM binnenkomt over de Transportstream Interface en deze weer -eventueel (gedeeltelijk) descrambled- weer verlaat, terug naar de host. Onder zien we de besturingscommunicatie tussen de Module en de Host over de Command Interface.

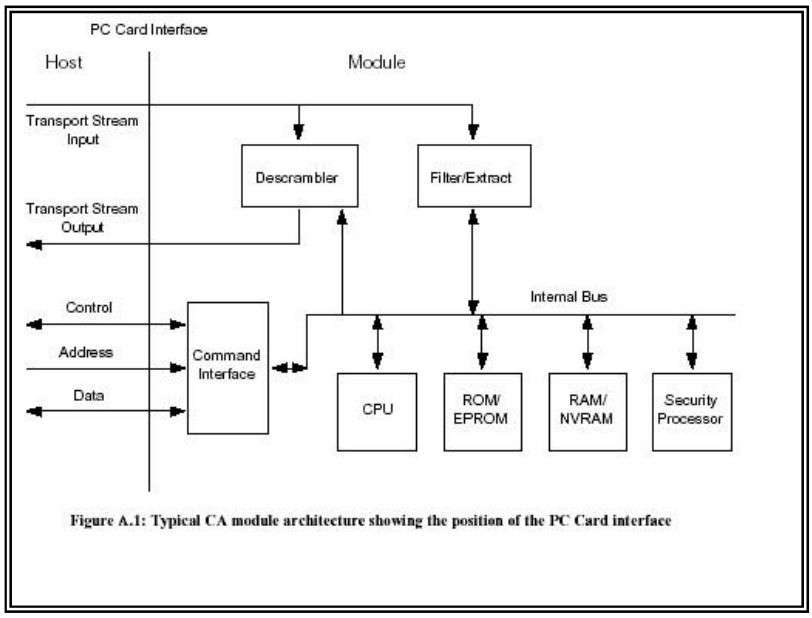


Figure A.1: Typical CA module architecture showing the position of the PC Card interface

Binnen de CAM vinden we onder meer de volgende componenten:

Descrambler	De descrambler decodeert selectief (delen van) de MPEG-2 transportstream. Dit gebeurt door de CPU periodiek controlwords te laten laden in de descrambler.
Filter/extract	Dit circuit filtert de besturingsinfo (ECM en EMM) uit de datastream.
CPU	Deze processor draait de applicatie(s) van het CA proces (onze firmware) en bestuurt de dataflow door de module en over de CI interface.
ROM/EPROM & RAM/NVRAM	In dit geheugen bevindt zich de programmatuur van het CA proces (onze firmware dus) en de hiervoor benodigde data (gegevens).
Security processor	Een aparte processor met een hogere beveiliging dan de CPU. Deze voert beveiligingsfuncties uit zoals decryptie, en beheert beveiligingsgegevens zoals keys, en entitlements. Hij kan zich in de module zelf bevinden, maar ook daarbuiten, op een aparte smartcard.

3.4 Algemene werking

Er wordt in EN50221 gesproken over host en module. Een **host** is het apparaat dat gebruik maakt van de module en betreft in ons geval dus de ontvanger. Een **module** is een apparaat dat niet zelfstandig werkt, maar in samenwerking met een host taken kan vervullen, in ons geval is dit dus de CAM.

De CI interface, het slot waar de CAM in de ontvanger wordt gestoken, is gedefinieerd op basis van het concept van **applicaties** die gebruik maken van **resources**. Een **applicatie** is een programma dat draait op een module en biedt de gebruiker aanvullende functionaliteit als aanvulling op de functionaliteit van de host (bijvoorbeeld descrambling). Een **resource** is een verzameling functionaliteit die zowel kan worden aangeboden door host als de module en die door een applicatie kan worden gebruikt. Enkele voorbeelden van resources zijn de MMI (Man machine interface), de Conditional Access Support en de Smart Card Reader Resource.

Communicatie tussen een applicatie en een resource vindt plaats in een session waarin objecten worden uitgewisseld die door de resource zijn gedefinieerd. Een **session** is een “gesprek” met een duidelijk begin en einde tussen een applicatie en een resource.

Een **object** is een bericht, bestaande uit een reeks bytes (tekens) die op een gestructureerde manier is ingedeeld: een **tag** (label) die aangeeft wat voor soort object het is, een lengteveld waarin staat hoeveel databytes er volgen en de daadwerkelijke databytes (de inhoud van het "bericht"). De uitwisseling van objecten binnen een session vindt plaats volgens een protocol. Een **protocol** is een voorgeschreven wijze waarop de objecten moeten worden uitgewisseld, zoals wie welke objecten mag verzenden en welk object een antwoord is op welk object. Zo zijn er bijvoorbeeld objecten om een session te openen en te sluiten.

De CI interface bestaat zoals we hebben gezien uit twee logische componenten: de **Transport Stream Interface** en de **Command Interface**. Beide logische componenten delen **dezelfde fysieke interface**: de PC-card interface.

- Over de **Transport Stream Interface** loopt de MPEG-2 transport stream in twee richtingen, de module in en uit.
- Over de **Command Interface** communiceren de host en de module hun commando's d.m.v. objecten.

In de MPEG-2 transportstream kunnen meerdere **PES** stromen (Packetized Elementary Stream) zitten welke bij elkaar gegroepeerd kunnen zijn in een **Service** en samen een televisieprogramma vormen (beeld, geluid, ondertiteling, etc.)

Als de MPEG-2 transportstream gescrambelde pakketten bevat én de module toegang kan verlenen tot deze service én de host heeft deze service geselecteerd, dan zullen de packets van deze service descrambled worden teruggegeven aan de host. Zo niet, dan worden de packets ongewijzigd retour gegeven. De module haalt zelf de voor descrambling benodigde elementen uit de transportstream zoals ECM en EMM messages.

Het is mogelijk om meerdere modules in één host te plaatsen. Het uitgangspunt van de CI interface is dat deze modules als het ware serieel geschakeld worden. De transportstream loopt de eerste module in en terug naar de host. Van de host naar de volgende module en weer terug. De transportstream wordt dus als het ware door alle modules geleid.

Het is aan te raden het bovenstaande nog eens aandachtig door te lezen. Een goed begrip van deze basiskennis is onmisbaar om de nu volgende beschrijving en EN50221 goed te kunnen begrijpen!

3.5 Objecten en Protocollen (Command Interface)

Alvorens de beschikbare resources in beeld te brengen waar we met onze applicatie(s) gebruik van kunnen maken, gaan we eerst nog wat dieper in op het principe van objecten en protocollen. Dit is uiteindelijk de taal waarin we met de ontvanger zullen gaan praten, dus zullen we dit tot op "bitniveau" moeten begrijpen.

Stel, we willen een session starten met de Resource Manager (zie 3.6.1 De Resource Manager). We stellen hiertoe volgens het hiervoor geldende protocol een **Open_session_request** object samen:

Open_session_request		
Open_session_request_tag	**Length field**	**Resource identifier**
91	04	00010041
soort object	*omdat er 4 bytes volgen*	*id van de Resource Manager, uit tabel*

Wat we dus verzenden aan de host is de volgende reeks van 6 bytes (hexadecimaal weergegeven): "**910400010041**".

Volgens het protocol gaat de host ons nu beantwoorden met **Open_Session_Response** object. Dit object gaat ons vertellen of de host een session met ons heeft geopend en zo ja wat het nummer van deze session is. Uiteindelijk kunnen en zullen we namelijk meerdere sessions tegelijkertijd geopend hebben. Wij ontvangen dus van de host bijvoorbeeld het volgende antwoord:

Open_session_response				
Open_session_response_tag	**Length field**	**Session status**	**Resource Identifier**	**Session_nb**
92	07	00	00010041	0001
Soort object	*7 bytes volgend*	*session opened*	*Id Resource Mngr*	*Sessienummer*

We krijgen dus de volgende reeks van 9 bytes terug van de host (hexadecimaal weergegeven): "**920700000100410001**"

Hierdoor weten we nu dat de session is geopend onder sessionnumber 0001. Nu zijn we dus in gesprek met de Application Manager!

Het veld Session status zou ook bijvoorbeeld F0 kunnen bevatten, in plaats van 00. In een tabel in EN50221 kunnen we de betekenis hiervan vinden: "*Session not opened, resource non existent*". Dan hebben we dus een probleem. Gelukkig is in een DVB ontvanger de Resource Manager altijd aanwezig en beschikbaar.

In Appendix 6 vinden we een complete log van het aanmeldingsproces van een CAM in een ontvanger.

Als we het bovenstaande goed begrijpen dan is het dus een kwestie van goed in kaart brengen welke resources er beschikbaar zijn en hoe we hiermee kunnen en communiceren: welke objecten zijn er beschikbaar en volgens welk protocol wisselen we deze objecten uit?

In de nu volgende paragrafen gaan we de standaard beschikbare resources en nog wat optioneel aanwezige resources bekijken. Zoals eerder aangegeven beperken we ons uit oogmerk van overzichtelijkheid tot een globale omschrijving. In EN50221 is een gedetailleerde beschrijving van ieder object, veld en protocol te vinden. Zoals gezegd, er valt echt niet te ontsnappen aan EN50221!

3.6 De Resources (Command Interface)

3.6.1 De Resource Manager

De **Resource Manager** is een resource op de host en vormt de manager van alle beschikbare resources op zowel de host als de aanwezige module(s). Er is een protocol van objecten waarmee de resource manager met de applications en de resource providers kan communiceren over beschikbare resources.

Het eerste dat een application of resource provider doet wanneer de module in de host wordt geplaatst, of wanneer de host wordt aangezet, is een session openen met de Resource Manager. De Resource Manager stuurt een **Profile Enquiry** object aan alle applications en/of resource providers die deze beantwoorden met een **Profile Reply** waarin de beschikbare resources (indien aanwezig) worden opgegeven. Nadat de Resource Manager alle resources in kaart heeft gebracht stuurt hij een **Profile Change** object aan alle applications en resource providers.

Na het ontvangen van het Profile Change object kan een application of resource provider, indien gewenst, met een **Profile Enquiry** object bij de Resource Manager informeren naar alle beschikbare resources. De Resource Manager antwoordt met een **Profile Reply** met alle beschikbare resources.

Pas na het ontvangen van het eerste Profile Change object staat het een application of resource provider vrij om sessions te openen of te accepteren. De oorspronkelijke session met de Resource Manager blijft open om eventuele wijzigingen in de beschikbare resources te kunnen ontvangen van de Resource Manager met een Profile Change object. Als de application of resource provider zelf een wijziging in de beschikbare resources wil doorgeven stuurt hij een Profile Change object naar de Resource Manager. Deze beantwoordt met een Profile Enquiry, waarop de application of resource provider een nieuwe Profile Reply zendt met de gewijzigde resource lijst. Indien dit de resourcelist in de Resource Manager wijzigt, dan wordt een Profile Change gestuurd aan alle applications en resource providers. Deze kunnen dan met een Profile Enquiry weer informeren naar de nieuwe resourcelijst.

3.6.2 Application Information Resource

De **Application Information Resource** is een resource van de host, vergelijkbaar met de Resource Manager. Waar de Resource Manager de manager van de Resources is, is de Application Information Resource de manager van de applications.

Iedere application zal na de Profile Enquiry initialisatie fase een session openen naar de Application Information Resource in de host. De host stuurt vervolgens een **Application Info Inquiry** object naar de application, die deze beantwoordt met een **Application Info** object. In dit object vinden we informatie zoals Application type (b.v. “01” = Common Access) en de toplevel (hoogste) menukeuze van de application. Deze menukeuze wordt door de host naar eigen inzicht opgenomen ergens in haar eigen menustructuur.

De session wordt open gehouden, zodat de host op ieder moment een **Enter Menu** object kan sturen waarna de application direct een MMI (Man Machine Interface) session zal starten met het hoofdmenu van de application. Dit gebeurt natuurlijk wanneer een gebruiker in de host naar de menustructuur van de host gaat, en hier de toplevel menukeuze van de application kiest.

3.6.3 Conditional Access Support resource

De Conditional Access Support is een resource op de host om Conditional Access applicaties te ondersteunen. Alle CA applications openen een session naar deze resource zodra de Application Info Inquiry is voldaan. De host verzendt een **CA Info Inquiry** object naar de application en de application beantwoordt deze met een **CA Info** object met informatie over de CA System ID's (Provider ID's) die de application ondersteunt. De host weet hierdoor welke application welk CA system kan decoderen. De session blijft vervolgens geopend om de host in staat te stellen gebruik te maken van deze ondersteuning door middel van de onderstaande CA PMT en CA PMT Reply objecten.

Als een gebruiker een programma selecteert, dan verstuurt de host aan één of meerdere CA applications een **CA PMT object** met hierin informatie over het geselecteerde programma, zoals de PES streams waaruit het programma bestaat en aanwijzingen hoe de ECM's gevonden kunnen worden. Als een gebruiker meerdere programma's heeft gekozen wordt voor ieder programma een CA PMT object verzonden. Het CA PMT bevat alle -maar ook alleen maar de- **CA descriptors** voor het geselecteerde programma uit de Program Map Table (PMT) in de MPEG-2 transportstream (zie: 3.8 MPEG-2 transportstream (Transport Interface))

De applications antwoorden met een **CA PMT Reply** object, waarna de host de application kan selecteren die de descrambling zal uitvoeren voor het geselecteerde programma. In het CA PMT Reply object is hiervoor het veld **ca_pmt_cmd_id** aanwezig. Hiermee geeft de host aan welke actie van de application wordt verwacht, zoals bijvoorbeeld direct descramble of dat eerst nog een MMI (Man Machine Interface) session moet worden gestart. Bijvoorbeeld om eerst kijkrechten aan te schaffen.

3.6.4 Host Control en Information Resources

Naast de bovenstaande belangrijke resources staan er nog een aantal andere resources ter beschikking.

Zo geeft de **DVB Host Control** resource de module de mogelijkheid om tijdelijk de besturing van de host over te nemen. Met het **Tune** object worden overgeschakeld naar een andere service (programma). In het object bevinden zich de hiervoor benodigde parameters (Network ID, Original Network ID, Transport Stream ID en service_id). Met **Replace, Clear Replace** en **Ask Replace** kan tijdelijk worden overgeschakeld (bijvoorbeeld voor een reclameboodschap).

Bij de **Date-Time resource** kunnen met het **Date-Time Enquiry** object de datum en tijd van de host worden opgevraagd.

3.6.5 Man Machine Interface Resource

Deze resource stelt de module in staat om met de gebruiker van de host te communiceren. Het geeft de module de controle over de display en kan toetsaanslagen van de gebruiker ontvangen. Een toepassing hiervan is vanzelfsprekend de menustructuur van onze CAM.

Er kan op twee manieren met de MMI resource worden gecommuniceerd: **Low-level MMI** en **High-level MMI**. Met Low-level MMI heeft de module de absolute controle over de graphics van de display en kunnen toetsaanslagen van de gebruiker rechtstreeks worden ontvangen. Bij High-level MMI zijn een aantal objecten gedefinieerd, waarin menu's en lijsten kunnen worden gecommuniceerd. De host bepaalt in dit geval de look-and-feel van de display.

Een MMI session wordt bijvoorbeeld opgestart als het **Enter menu** object in de session met de Application Information Resource wordt ontvangen (zie 3.6.2 Application Information Resource). Een MMI session kan door de host en de module worden beëindigd door verzending van het **Close_MMI** object. Om menu's tussen applications te kunnen laten wisselen zonder het "flitsen" van het "onderliggende" programma, kan een **delay** parameter worden meegegeven.

Met het **Display control** object kunnen zowel de karakteristieken van de display worden opgevraagd als de gewenste modus worden ingesteld. Er zijn drie mogelijkheden: Bitmap graphics door de huidige uitzending, Bit map graphics in plaats van de huidige uitzending of character based High-level modus. Gezien de complexiteit van de Low-level modus en onze doelstelling beperken wij ons hier tot de eenvoudigste: de character based high-level modus.

Op het display control object antwoordt de host met een **Display reply** object. Voor onze high-level mode is in het reply object alleen de tabel met ondersteunde karakterset van enig belang en de bevestiging van de geselecteerde modus.

In High-level MMI kan met het **Text** object een blok tekst door de module naar de host worden verzonden om op de display te worden getoond. Het is mogelijk hier wat controlcodes in op te nemen om de tekst enigszins op te maken. Met het **Enq** object kan een vraag op de display worden gesteld. Het door de gebruiker ingegeven antwoord wordt door de host met een **Answ** object teruggestuurd. Hierin bevindt zich de parameter **answ_id**, die de waarde 00 heeft, als de gebruiker op cancel heeft gedrukt.

Met het **Menu** object kan een menu op het scherm worden gezet waaruit de gebruiker een keuze kan maken. In het object kunnen en menutitel, -subtitel en -ondertitel worden opgenomen, tezamen met een aantal keuzes. In deze High-level MMI mode bepaalt de host natuurlijk hoe dit menu wordt getoond en op welke wijze de gebruiker een keuze kan maken. De gemaakte keuze wordt met een Answ object retour gezonden. Hierin bevindt zich het veld **Choice_ref**, die de waarde 01 heeft voor de eerste keuze, de waarde 02 voor de tweede keuze, etc. Een waarde 00 geeft aan dat de gebruiker het menu zonder keuze heeft beëindigd (escape).

Het kan echter ook wenselijk zijn een lijst af te drukken, zonder dat hier een keuze uit hoeft of kan worden gemaakt. Dit is mogelijk met het List object, dat een bijna gelijke structuur heeft als het Menu object, maar dan zonder Choice_ref veld natuurlijk.

3.6.6 Overige resources

Daarnaast is er nog een aantal andere resources die we hier voor de volledigheid kort in beeld brengen. Als we deze in het kader van onze studie nodig blijken te hebben, zullen we ze later verder uitwerken.

Zo is er de **Low-speed communication class** die ons in staat stelt om bijvoorbeeld over een modem of kabel systeem te communiceren. Met de optionele **Authentication resource** kan een autorisatie worden geregeld, waardoor een module alleen bevoegd is bepaalde signalen te verwerken. Met de **EBU Teletext display resource** kan informatie via de teletekst functie van de host op de display worden afgedrukt. De **Smart Card Reader Resource class** kan zowel door door de host beschikbaar worden gesteld als door (een andere) module. Met de objecten **Smart Card Cmd** en **Smart Card Reply** kunnen beheer- en antwoord commando's worden uitgewisseld. **Smart Card Send** en **Smart Card Reply** verzenden en ontvangen gegevens van de smart card. Met de optionele **DVB EPG Future Event Support Class** wordt het mogelijk voor een EPG (Electronic Program Guide) applicatie in de host te communiceren met een CA applicatie. Zo kan de host in de EPG lijst van de komende programma's direct tonen of de kijker gerechtigd is het programma te zien, of dat hier bijvoorbeeld eerst een aparte handeling (zoals aankoop van kijkrechten) voor moet plaatsvinden.

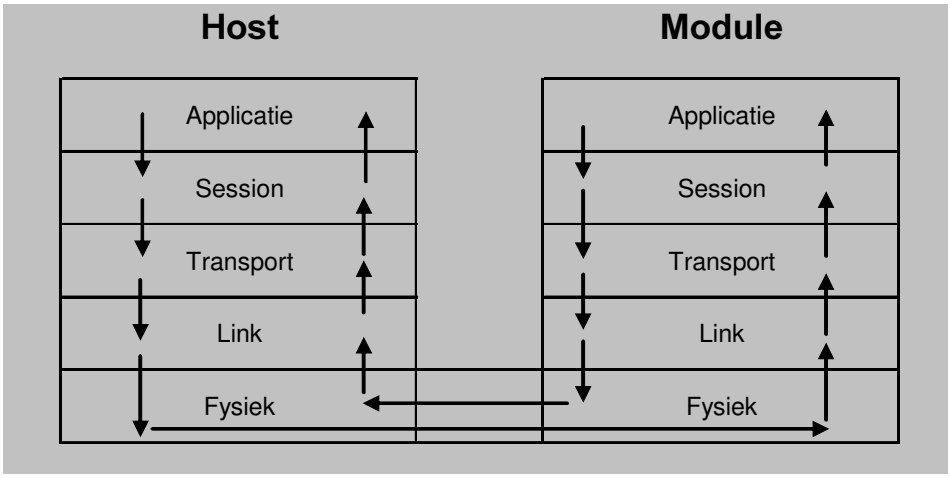
3.7 Communicatie tussen Host en Module

3.7.1 Het principe van layering

Je zou gezien de algemene beschrijving denken dat een application rechtstreeks “praat” met de resource. Een application zou dan een object (de reeks tekens) samenstellen en dit rechtstreeks zenden aan de resource. De resource zou de reeks tekens inlezen en reageren. Helaas is de werkelijkheid natuurlijk weer ingewikkelder. De communicatie moet namelijk uiteindelijk over een fysieke interface (de pc-card interface) naar de andere kant en hier komt een transportmechanisme om de hoek kijken, met componenten zoals buffers, registers, spanningswisselingen op pinnen, etc. Het is onder andere hierom dat tussen de application en de resource in zowel de host als de module een aantal “lagen”, de zogenaamde **layers,** zitten. Aan de verzendende kant praat iedere layer met de eronder liggende layer, die de boodschap weer doorgeeft aan de layer daaronder tot uiteindelijk de fysieke layer is bereikt. Hier wordt de data via de pinnen van de interface naar de andere kant “overgeseind”. Aan de andere kant wordt de boodschap weer door de fysieke layer aan de layer erboven doorgegeven, die deze boodschap weer aan de layer daarboven doorgeeft, tot deze bij de bestemming (de applicatie of de resource) aankomt. Zij die met het zogenaamde *OSI model* op de hoogte zijn, zullen geen moeite met dit concept hebben.

Dit veel toegepaste gelaagde transportmechanisme zorgt ervoor dat bijvoorbeeld een internet applicatie kan worden ontwikkeld zonder kennis van een specifieke netwerkkaart. Een specifieke laag, bijvoorbeeld een driver voor een netwerkkaart, kan worden ontwikkeld door mensen met de hiervoor benodigde specifieke kennis. De laag er bovenop, die bijvoorbeeld een transportconnectie opent met een andere computer op internet, kan worden ontwikkeld zonder specifieke kennis van netwerkkaarten en drivers hiervoor. Als je maar weet hoe je de driver moet aanroepen met de juiste parameters om de data over te dragen. Voordeel is ook dat bijvoorbeeld de onderste laag, de netwerkkaart, makkelijk kan worden vervangen door een andere kaart, met een andere driver, mits die maar op dezelfde wijze kan worden aangeroepen. De applicatie op het hogere niveau hoeft hiervoor dus niet te worden aangepast. Dit biedt dus de nodige flexibiliteit. Nog een ander voordeel is dat wanneer op een lager niveau reeds een transportconnectie met een andere computer is gemaakt, een andere applicatie die ook wil communiceren met een applicatie op die andere computer, er gebruik kan worden gemaakt van dezelfde transportconnectie. Er behoeft dan geen nieuwe transportconnectie te worden gemaakt.

In feite communiceert iedere layer met zijn soortgenoot aan de andere kant. Een layer accepteert een object van een bovenliggende layer, stopt deze objecten in zijn eigen object(en), voegt adressering toe voor de corresponderende layer aan de andere kant en draagt het object over aan de layer eronder. Ook voor de communicatie tussen de host en de module zijn (erg) globaal op deze wijze layers gedefinieerd:



3.7.2 De layers

De in de voorafgaande paragraaf zeer globaal benoemde layers worden nu verder uitgewerkt. Voor de Transport Stream Interface en de Command Interface zijn namelijk aparte layers gedefinieerd:

<table>
<tr><th>Transport Stream Interface</th><th colspan="4">Command Interface</th></tr>
<tr><td rowspan="2">Upper layers

(MPEG-2 specs ISO 13818)</td><td colspan="4">Application layer
Resources</td></tr>
<tr><td>User interface</td><td>Low speed communications</td><td>System</td><td>Optional extensions</td></tr>
<tr><td></td><td colspan="4">Session layer</td></tr>
<tr><td rowspan="2">Transport Layer
(MPEG-2 specs ISO 13818)</td><td colspan="4">Generic transport sublayer</td></tr>
<tr><td colspan="4">PC Card transport sublayer</td></tr>
<tr><td colspan="5">PC Card Link layer</td></tr>
<tr><td colspan="5">PC Card physical layer</td></tr>
</table>

1. De application verzoekt de session layer om een nieuwe session te openen met een resource. Nadat de session door de session layer is geopend, kan de application beginnen de te verzenden data in objecten te stoppen: (**Application Protocol Data Unit = APDU**).
2. De applicatie geeft de APDU's door aan de session layer.
3. De session layer stopt één of meer van deze APDU's in een **Session Protocol Data Unit (SPDU)** en draagt deze SPDU's over aan de Transport Layer.
4. De transport layer voegt één of meerdere SPDU's in één of meer **Transport Protocol Data Units (TPDU)** samen en draagt deze over aan de PC-card link layer.
5. Op de PC-card Link Layer wordt de TPDU's ingedeeld in **Link Protocol Data Units (LPDU)** die zijn afgestemd op de in de Physical Layer beschikbare buffer. Hierbij kunnen TPDU's van meerdere transport connections door elkaar worden verzonden. Een Link Connection wordt automatisch tijdens de initialisatie van host of module tot stand gebracht.
6. Op de Physical Layer wordt de data daadwerkelijk overgebracht naar het andere systeem. De bits worden door middel van wisseling van elektrische spanning op verschillende pinnen met een bepaalde snelheid gezet en gelezen door het andere systeem. De specificaties van dit niveau zijn dan ook fysiek van aard, zoals betekenis van pinnen, spanningshoogte, bitrate, etc. In ons geval betreft dit dus de specificaties van een PC-card (PCM CIA module).
7. In de host loopt deze informatiestroom weer naar boven via de Physical Layer, de Link Layer, de Session Layer naar de resource. Het antwoord volgt natuurlijk weer de omgekeerde weg.

Het spreekt bijna vanzelf dat op elke layer een protocol is benoemd volgens welke deze layer communiceert met de corresponderende layer aan de andere kant. Zo moeten bijvoorbeeld een session en een transport connection kunnen worden geopend en gesloten. Daar er op enig moment meerdere sessions en meerdere transport connections geopend kunnen zijn, zullen er identificaties moeten zijn om deze stromen "uit elkaar" te houden. Een session wordt daarom geïdentificeerd met een **Sessionnumber** en een transport connection met een **Transport Connection Identifier**.

Nu we deze problematiek globaal in kaart hebben, gaan we de objecten en de protocollen per layer nader bekijken. Voor ons doel, het ontwikkelen van software voor de CAM, is een goed en nauwkeurig begrip van deze communicatie natuurlijk van essentieel belang.

3.7.3 PC-card layers

De specificaties van deze onderste lagen komen overeen met de standaard definities voor een PC-Card interface (PCM-CIA module). Op het onderste niveau, de Physical Layer, worden zaken gedefinieerd als pinbezetting, spanning, data transfer rates, etc. In Appendix A van EN50221 vinden we de specificaties hiervan. Op deze Physical Layer worden de gegevens daadwerkelijk fysiek overgedragen met spanningswisselingen op de pinnen naar de ontvangende partij.

De PC-Card Link Layer heeft van het bovenliggende Transport Layer TPDU objecten ontvangen ter verzending en maakt hiervoor LPDU objecten, afgestemd op de bovenvermelde beschikbare buffer op de Physical Layer. Een Link connection komt automatisch tot stand bij de initialisatie die plaatsvindt zodra er op fysiek niveau contact is gemaakt. Bij deze initialisatie wordt de Card Information Structure gelezen en wordt de card in de juiste modus geconfigureerd. Hierbij wordt ook de buffergrootte onderhandeld. Ieder LPDU object bestaat uit een header van twee bytes en een deel van een TPDU, het geheel niet groter dan de buffergrootte. De eerste byte is het nummer van de transport connection waartoe het deel van de TPDU behoort. Het tweede byte heeft in het most significant bit (linker bitje) een “1” als er nog meer fragmenten van de TPDU volgen en een “0” als dit niet zo is. De andere 7 bits zijn gereserveerd en hebben de waarde “0”. Elke LPDU heeft maar de gegevens van één (deel van een) TPDU aan boord. Als er meerdere transport connecties tegelijkertijd over de link Layer lopen, dan worden fragmenten van beide TPDU’s door elkaar verzonden om een eerlijke verdeling van de beschikbare bandbreedte te verkrijgen.

3.6.4 Generic Transport Layer

De functie van de Transport Layer is de door de hogere Session Layer aangeleverde Session Protocol Data Units te transporteren naar de Transport Layer aan de andere kant, en de van de andere kant ontvangen Session Protocol Data units aan de eigen Session Layer aan te leveren. De Transport layer is dus het transport mechanisme van een session.

Communicatie op Transport Layer nivo vindt plaats volgens uitwisseling van **Command Objects**. De host verzendt een Command Transport Protocol Data Unit **C_PTDU**. De module antwoordt met een Response Transport Protocol Data Unit **R_PTDU**. De module kan geen transport beginnen en moet wachten tot de host begint. Er zijn in totaal 11 soorten **Transport Layer objects**, waarvan sommigen alleen door de host kunnen worden verzonden, sommigen alleen door de module en sommigen door beiden.

Een C_PTDU van de host bevat slechts één Transport Protocol object. Een R_PTDU van de module kan één of twee Transport Protocol Objecten bevatten. Het enige of het tweede object in een R_PTDU van de module is altijd een status object (T_SB).

Iedere transport connection heeft een **transport connection identifier** van 1 byte. Omdat 0 is gereserveerd, kan de host over alle modules tegelijkertijd 255 connections geopend hebben. Volgens EN50221 specificaties minimaal 16 per module, maar het liefst 255 verdeeld over de in de host aanwezige modules. De identifier wordt bepaald door de host.

De Objects

Nu gaan we de 11 soorten objecten bekijken en het protocol volgens welke ze worden uitgewisseld.

Transport Layer objects	
1. Create_T_C	Deze creëert een nieuwe connectie en bevat de connection identifier. (Alleen door host).
2. C_T_C_Reply	Antwoord hierop van de module met de connection identifier
3. Delete_T_C	Wist een transport connectie en bevat als parameter de te wissen connectie. Host en module kunnen hem verzenden, maar module alleen als antwoord op een poll (*) of data van de host
4. D_T_C_Reply	Het antwoord hierop. Omdat dit soms niet aankomt heeft Delete_T_C een timeout. Als die is bereikt kunnen de acties worden genomen die normaal o.g.v. de reply worden genomen.
5. Request_T_C	Een verzoek aan de host om een nieuwe transport connectie. Wordt met een bestaand connectienummer verzonden en alleen als antwoord op een poll (*) of data van de host.
6. New_T_C	Antwoord op Request_T_C. Wordt dezelfde connectie verzonden als de request en bevat de nieuwe connection identifier. New_T_C wordt direct gevolgd door een Create_T_C om de nieuwe connection te maken.
7. T_C_Error	Wordt verzonden met daarin 1 byte met het errornummer. In deze versie wordt deze alleen als antwoord op een Request_T_C verzonden dat er geen connecties meer mogelijk zijn.
8. T_SB	Wordt als antwoord op alle objecten van de host verzonden, eraan vast geplakt of apart en geeft in één byte aan of de module nog data te verzenden heeft
9. T_RCV	Wordt verzonden door host als antwoord op een T_SB met het verzoek de data aan de host te sturen.
10. T_Data_more 11. T_Data_last	Deze bevatten de daadwerkelijke data van of naar de module. De module mag allen verzenden op verzoek met een T_RCV. T_data_more wordt gebruikt als een Protocol Data Unit van een hoger nivo moet worden gesplitst omdat het te groot is om in één object te worden verzonden (door externe beperking). T_data_last bevat het laatste fragment van een gesplitste PDU of het enige fragment van een PDU. Bij meerdere segmenten, moet ieder datapacket wachten op een aparte T_RCV.

(*) Pollen gebeurt met een lege "T_Data_last"

Het protocol

Als we de objecten goed bestuderen, dan ligt het protocol voor de uitwisseling van deze objecten min of meer voor de hand.

Als een host een transport connection wil starten naar een module (en alleen de host kan beginnen!) dan stuurt de host een Create_T_C object naar de module. De module antwoordt direct met een C_T_C_Reply. Als deze reply niet binnen een time-out periode wordt ontvangen dan is de transport connecion mislukt, en zal het transport connectionnumber opnieuw worden gebruikt, bijvoorbeeld om opnieuw een connectie te proberen op te bouwen met de module.

Na de C_T_C_reply kan de host data aan de module gaan verzenden met T_data_last (de host verstuurt altijd maar één Protocol Data Object). Is er geen data te verzenden, dan poll't de host de module of die data te verzenden heeft met een leeg T_data_last object. De module antwoordt de host met een T_sb object waarin met één byte wordt aangegeven of de module data te verzenden heeft. Is dit het geval, dan antwoordt de host met een T_RCV object waarop de module met T_data_more of T_data_last de data verzendt. Indien er meerdere data objecten te verzenden zijn, dan wordt de data verzonden met T_data_more. Het laatste pakket wordt verzonden met T_data_last. Voor iedere T_data_more of T_data_last moet de module wachten op een T_RCV van de host.

Indien een module een nieuwe transport connection wil starten dan kan het antwoord van de module ook een Request_T_C zijn waarop de host antwoord met een T_C_error als er geen connections meer beschikbaar zijn, of een New_T_C met het nieuwe connectionnumber. De module antwoordt hierop met een C_T_C_reply, en na het ontvangen van de T_RCV van de host kan de module de data over deze connectie weer gaan verzenden met T_data_more en T_Data_last.

Zowel de host als de module kan vervolgens de connection sluiten door het verzenden van een Delete_T_C object en na ontvangst van de bevestiging hiervan met een D_T_C_Reply object is de connection gesloten.

3.7.5 Session Layer

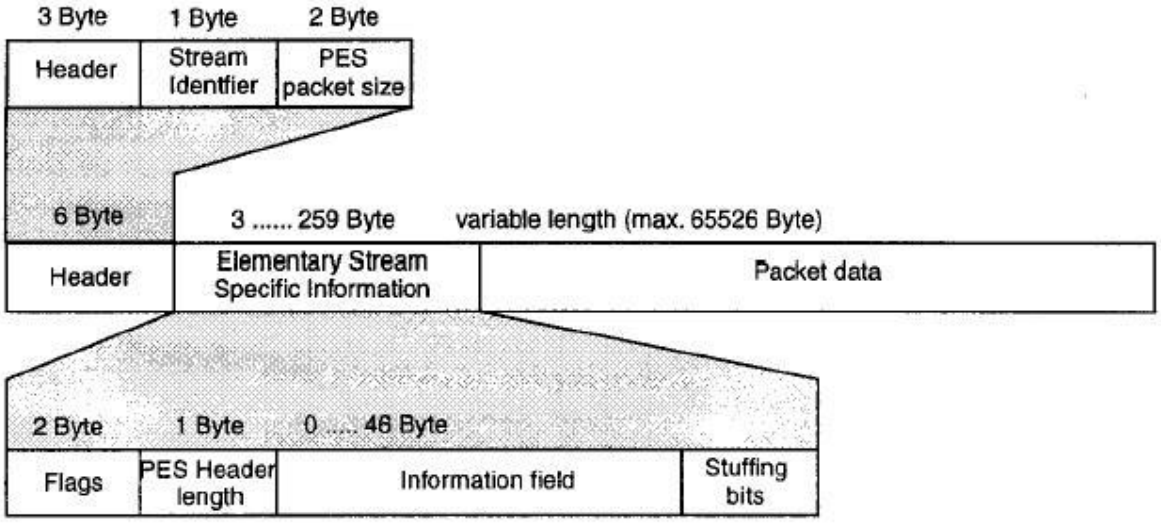
De session layer voorziet in het mechanisme waarmee applications gebruik kunnen maken van de resources. Een resource kan zich op de host of de module bevinden. Een module kan ook via de host gebruik maken van een resource op een andere module.

Sommige resources kunnen meer dan één sessie aan sommigen niet. Zo kan bijvoorbeeld de display wellicht slechts één session aan. Heeft de display windows, dan kunnen meer sessies worden geopend, anders niet. Dan wordt een Resource_busy reply object verzonden. De volgende objecten zijn gedefinieerd op de sessionlayer.

Open_session_request	Wordt verzonden door een applicatie over zijn transport connection met verzoek om gebruik van een resource. De host kan de resource direct beschikbaar stellen of een nieuwe transport connection maken naar een module die de resource beschikbaar heeft.
Open_session_response	Dit verzendt de host naar de applicatie om een sessienummer toe te kennen of een melding dat de resource niet beschikbaar is.
Create_session	Wordt door een Host verstuurd aan een module die een resource beschikbaar heeft om aan een session_request van een andere module te kunnen voldoen over een nieuwe transport connection.
Create_session_response	Is het antwoord van een resource provider in een module aan de host zodat de host aan de aanvragende module kan aangeven of de session kan worden geopend.
Close_session_request	Wordt door host of module verzonden om een session te beëindigen
Close_session_response	Wordt door host of module verzonden om de beëindiging van een session te bevestigen.
Session_number	Een Session_number gaat altijd vooraf aan een Session Protocol Data Unit (SPDU) die een Application Protocol Data Unit (Application Protocol Data Unit) bevat.
Session_nb(n, data)	De daadwerkelijke data, voorafgegaan door het session number.

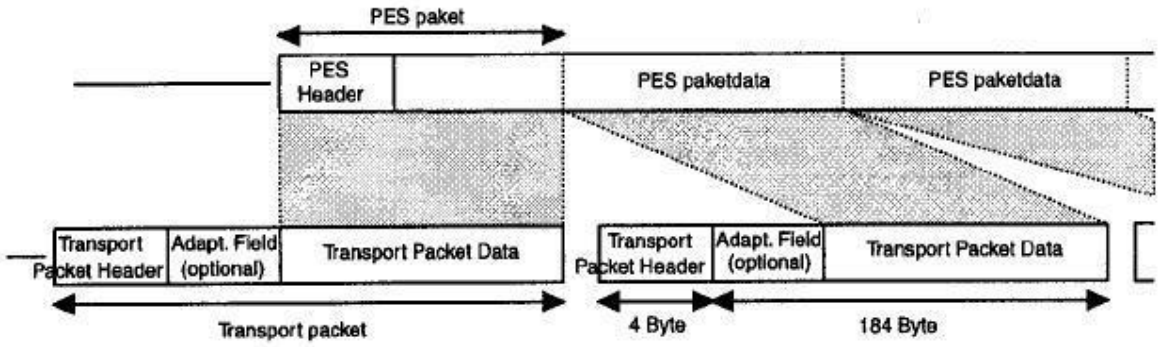
3.8 MPEG-2 transportstream (Transport Interface)

Zoals we hebben gezien communiceren de *applicaties* en de *resources* op de module over de *Command Interface*. De **MPEG-2 transportstream** komt echter over de *Transport Interface* de module binnen en verlaat deze ook weer via de transport interface. De MPEG-2 transportstream bestaat uit meerdere **Packetized Elementary Streams (PES),** en iedere PES bevat één elementaire stroom data, zoals beeld, geluid, ondertiteling, besturingsinformatie, etc. Een PES stroom zelf bestaat uit elkaar opvolgende packets met een (lange) variabele lengte. Elk packet begint met een "header" van 6 bytes. Het volgende dataveld bevat hoofdzakelijk stuurinformatie (Elementary Stream Specific Information) en is maximaal 259 bytes. Als laatste volgt de eigenlijke data met een variabele lengte van maximaal 65526 bytes. Het eerste veld, Flags, bevat statusbits voor bijvoorbeeld scrambling, copywright, etc.



Structuur van de PES-stroom.

De **MPEG-2 transport stream** is opgebouwd uit transport packets met een (korte) vaste lengte van 188 bytes. De **transport packet header** is 4 bytes lang. De overige 184 bytes zijn voor de daadwerkelijke data. De PES packets worden in stukjes gehakt en verdeeld over meerdere transport packets. In onderstaand figuur wordt de relatie weergegeven tussen de transportstream en de PES stream.



Samenhangende structuur van transportstroom en PES-stroom.

De PES header volgt altijd direct de Transport header tenzij een 'adaptation field' wordt meegezonden. In dat geval volgt de PES header direct daarna. Aan het einde van een PES packet kunnen een aantal 'stuffing bytes worden toegevoegd, die geen informatie bevatten.Een PES wordt geïdentificeerd met een

PID nummer, dus PES packets met hetzelfde PID nummer behoren tot dezelfde PES stroom. Hierdoor kan de ontvanger de verschillende PES stromen uit elkaar houden en selecteren. De transport packet header bestaat uit 32 bits (4 bytes). De PID vinden we hierin op bit 12 tot en met bit 24 (13 bits).

Verschillende PES stromen vormen samen een programma, geïdentificeerd met een programmanummer van 2 bytes. Voor informatie over programma's wordt de **Program Specific Information (PSI)** gebruikt. Er zijn in de PSI een aantal tabellen gedefinieerd, waarvan onderstaand de belangrijkste:

Table	Locatie	Inhoud
Program Association Table (PAT)	PID = 0	Voor ieder programmanummer in de transportstream de PID van de *Program Map Table* van het programma. (Daarnaast in programma 0 de PID voor de Network Information Table)
Program Map Table (PMT)	PID per PMT	Voor ieder programma in de transportstream de PID nummers en descriptors van het programma.
Network Information Table (NIT)	Programma 0 in PAT	Karakteristieken van het transmissienetwerk zoals frequentieband, centrale frequentie, kanaalbandbreedte, transponder, etc.
Conditional Access Table (CAT)	PID = 1	Geeft voor iedere CA provider de PID voor de EMM messages en eventuele parameters.

Een ontvanger haalt uit de Program Association Table (PID=0) de programma's die zich in de transportstream bevinden. In het geval dat de gebruiker programma x kiest, kan de ontvanger uit de Program Association Table bij programma(x) de PID halen voor de Program Map Table van dit programma. In deze Program Map Table vindt de ontvanger de PID's waaruit het programma bestaat (beeld, geluid, etc.) en de eventuele descriptors van het programma, zoals de CA-descriptor benodigd voor descrambling.

Een descriptor is een reeks tekens die informatie over het betreffende onderwerp bevat en we komen op verschillende plekken verschillende soorten descriptors tegen. De CA-descriptor bijvoorbeeld bevat de informatie die we nodig hebben om de ECM's en EMM's te vinden. De ECM's en EMM's worden namelijk met aparte PID nummers verzonden. We kunnen een CA descriptor op twee plaatsen tegenkomen:

- In de **Program Map Table** geeft de CA-descriptor de PID van de **ECM**'s van een programma;
- In de **Conditional Access Table** geeft de CA-descriptor de PID van de **EMM**'s van een provider.

Naast de door MPEG-2 gedefinieerde tabellen zin er in DVB nog een aantal extra tabellen gedefinieerd zoals bijvoorbeeld de **Bouquet Associaton Table (BAT)** en de **Event Information Table (EIT)**. Voor iedere tabel is een PID gedefinieerd waarmee deze wordt verzonden, waarbij sommige tabellen met dezelfde PID worden verzonden. Het onderscheid tussen de verschillende tabellen wordt gemaakt met een **Table_id**. Geen van de tabellen mag scrambled worden verzonden, op de EIT (Event Information Table) na, maar de data mag wel scrambled worden verzonden. Er vindt dan meestal 'stuffing' plaats om de overgang tussen gescrambelde en niet gescrambelde data te laten plaatsvinden.

Eén van onze uitdagingen zal zijn de ECM's te filteren en de hierin verstopte controlwords te vinden. De structuur van ECM's en EMM's is in ISO13818-1 slechts globaal voorgeschreven in een algemeen PES packet format:

Syntax	Aantal bits	Mnemonic
Packet_start_code_prefix	24	Bslbf
Stream_id	8	Uimsbf
PES_packet_length	16	Uimsbf
Databytes (aantal = PES_packet_length)	8	bslbf

- **packet_start_code_prefix**: The packet_start_code_prefix is een 24-bits code. Samen met de stream_id erna, vormt het een packet start code die het begin van een packet identificeert. De packet_start_code_prefix is de volgende reeks bits: '**0000 0000 0000 0000 0000 0001**' (0x000001).
- **Stream_id:** geeft de soort data weer in het packet en hiervoor is in ISO13818.1 een tabel gedefinieerd. Bij een ECM zal deze waarde '**1111 0000**' (0xF0) zijn en bij een EMM '**1111 0001**' (0xF1).
- **PES_packet_length**: Een 16 bits veld dat het aantal bytes in het PES packet aangeeft dat na dit veld volgt. (De waarde 0 geeft aan dat de PES Packet length niet is gespecificeerd en niet begrensd is. Dit kan alleen vookomen in PES packets waarvan de data een video elementary stream is die in Transport Stream packets is verzonden.)

Voor de databytes in een ECM/EMM wordt in ISO13818 geen structuur voorgeschreven. Dit zal waarschijnlijk het onderzoeksonderwerp worden bij de bestudering van de diverse coderingen zoals Irdeto, Seca, etc.

Deze informatie over de transportstream is afkomstig uit de ISO13818.1 specificaties. Er is hier slechts een globale beschrijving met enkele details opgenomen om de werking te doorgronden en een beeld te vormen van wat ons te wachten staat. Voor de details van alle packets, descriptors, tables, etc. kan ISO13818-1 worden geraadpleegd.

3.9 En nu verder…

We weten nu alle benodigde details van de werking van een CAM om hiervoor firmware te ontwikkelen. We weten wat we moeten doen om ons aan te melden in de ontvanger en welk communicatieproces ons hierbij te wachten staat. Ook weten we hoe we een menstructuur kunnen aanbrengen en ook hoe de ontvanger ons gaat verzoeken bepaalde programma's te descrambelen. We hebben gezien hoe de opbouw van de transportstream is en hoe we daar de voor de descramble benodigde gegevens in kunnen vinden. Het wordt nu tijd dat we ons gaan verdiepen in één specifieke CAM en de manier waarop we hiervoor software kunnen ontwikkelen.

Literatuurverwijzingen:

- http://www.bjpace.com.cn/data/tec/tec-DVB/DVB%20BlueBooks%20Standards/Specifications%20and%20Standards/interfacing/dvb-ci/EN50221.PDF
- http://neuron2.net/library/mpeg2/iso138181.doc
- http://users.pandora.be/satelliet/mpegnorm.pdf

4. Matrix Cam

4.1 Inleiding

De op het moment van het schrijven van dit rapport meest voorkomende UCAS Cams zijn de Matrix in al zijn generaties, de Xcam en de Dragon. Algemene info over de beschikbare CAM's is eenvoudig op internet te vinden. De Matrix CAM kennen we tot op heden in vier generaties: De Matrix, de Matrix Reloaded, de Matrix Revolutions en de Matrix Reborn. Gezien de geboortefrequentie van nieuwe versies van CAM's kunnen er dus best op het moment dat je dit leest al weer nieuwe varianten zijn verschenen.

We nemen de Matrix CAM als onderzoeksonderwerp om twee redenen. De eerste is dat de schrijver toevallig over drie van deze CAM's beschikt: de Reloaded (embedded), de Revolutions en de Reborn (de laatste twee geleend van Bommeltje ☺). De tweede reden is dat er de meeste documentatie en zelfs broncode voor te vinden is.

4.2 De componenten

Op de thuispagina van de Matrix cam vinden we de specificaties van de (eerste generatie) Matrix CAM. Specificaties van de Reloaded en de Revolutions zijn helaas niet vermeld, maar ons uitgangspunt moet maar zijn dat de (eventuele) verschillen niet van invloed op onze firmware zullen zijn. De drie voor ons doel meest interessante processoren van de Matrix Cam zijn:

1. **Sidsa's MACtsp Multimedia engine**: De MPEG-2 transportstream loopt door deze door de firma Sidsa geproduceerde engine en kan hiermee worden gelezen en bewerkt. Met de MACtsp kunnen de ECM's en EMM's worden gefilterd en kunnen (delen van) de MPEG-2 transportstream worden descrambled. Het hart van de Sidsa MACtsp wordt gevormd door een ARM (ARM7TDMITM) processor. Dit is de processor (CPU) waarop we onze firmware zullen ontwikkelen. De MACtsp is dus eigenlijk de CPU, de Filter/Extract, Security processor en Descrambler uit ons plaatje in 3.3 De structuur.
2. **Flash memory**: een in aparte blokken ingedeeld flashmemory. Hierin kunnen we onze firmware laden.
3. **Xilinx:** De Xilinx verzorgt het MPEG transport tussen de PCMCIA interface en (ik denk) de Sidsa MACtsp..

Verder zijn er nog extra geheugenchips aanwezig op de cam, zoals bijvoorbeeld de CMOS SRAM. Het is de schrijver niet gelukt verdere details over de functie hiervan in de CAM te verkrijgen. In ieder geval heeft iedere computer gestuurde apparatuur een RAM nodig voor een (snelle) buffer voor het uitvoeren van commando's en het draaien van programma's.

In Appendix 4: Specificaties Matrix zijn meer technische details opgenomen van de diverse onderdelen van de CAM. Gezien onze doelstelling en de beperkte hardwarekennis van de schrijver gaan we op deze onderdelen niet dieper in dan waarschijnlijk voor het ontwikkelen van firmware noodzakelijk is. Voor de liefhebbers zijn op de Matrix homepage diverse verwijzingen te vinden naar de datasheets van deze processoren.

Het enorm goede nieuws voor ons is dat voor de ARM processor complete SDK's (Software Development Kit) beschikbaar zijn, waarmee we de firmware kunnen ontwikkelen. En ons geluk houdt hier niet mee op. Er zijn namelijk ook nog eens standaard routines voor de MACtsp beschikbaar (core libraries) waarin reeds standaard routines zijn opgenomen voor de taken die ons staan te wachten, zoals het filteren van ECM's en EMM's uit de transportstream, het aan de hand van controlwords descrambelen van (delen van) de transportstream en communicatie met een smartcard. Met de Development Kit, de beschikbare documentatie hiervan, de core libraries en de specificaties EN50221 en ISO13818-1 hebben we dus een volle gereedschapskist voor de ontwikkeling van onze firmware!

4.3 Sidsa's MACtsp: core libraries

De Sidsa MACtsp multimedia engine bevat dus het grootste deel van de functionaliteit van de CAM. Hierin bevinden zich dan ook de voor onze firmware belangrijkste onderdelen. Het hart van de multimedia engine wordt gevormd door de ARM7 processor. Voor deze processor zijn verschillende ontwikkelomgevingen (SDK's) te vinden waarmee je zowel Assembly code en C code kunt schrijven en compileren.

Sidsa heeft voor de MACtsp core libraries ontwikkeld waarin een aantal functies zijn geprogrammeerd die wij vanuit onze eigen firmware kunnen aanroepen. Zo zijn er standaard routines om ECM/EMM berichten te filteren, om controlwords in de descrambler te laden, etc. In dat opzicht is ons leven dus toch wat eenvoudiger dan wij ons misschien op basis van de voorgaande hoofdstukken hadden voorgesteld. Zonder de basiskennis uit de voorgaande hoofdstukken zouden we ons echter geen raad weten met de aangeleverde routines.

Voor de in het vorige hoofdstuk beschreven layers vinden we in de core libraries aanroepbare routines (API calls). Hoewel het goed begrijpen van deze functies nog verdere studie zal vergen, geven ze ons wel een behoorlijk gevoel bij de firmware die we gaan ontwikkelen. Het is hierom dat we ze in dit hoofdstuk globaal doornemen.

De informatie over de API calls in de onderstaande tabellen is te vinden in de diverse .doc bestanden die bij de core libraries worden meegeleverd. In de header van iedere tabel is het Word document vermeld waarin de gedetailleerde informatie kan worden gevonden. Het betreft hier API (Application Program Interface) calls in de programmeertaal C.

4.3.1 Configuratie routines (config.doc)

In de board.c file kunnen we een aantal parameters definiëren waarmee de configuratie van de CAM zal plaatsvinden. Tevens kunnen we aangeven of de pcmciaInit functie van de link Layer automatisch willen laten aanroepen tijdens de boot van de CAM.

4.3.2 PCMCIA interface (pcmcia.doc)

Deze functies kunnen worden aangeroepen voor het verzenden en ontvangen van de LPDU's (Link Protocol Data Units), zoals beschreven in hoofdstuk 3.

Initialisatie	
PcmciaInit	Initialiseert de pcmcia interface.
Interface handling	
pcmciaOpen	Opent de pcmcia interface en onderhandelt de bufferlengte.
pcmciaClose	Sluit de pcmcia interface.
pcmciaWrite	Stuurt een LPDU naar de pcmcia interface.
pcmciaRead	Leest een LPDU van de pcmcia interface.
pcmciaDataReady	Controleer of er data te ontvangen is (True or False).
pcmciaBufferLengthGet	Retourneert de onderhandelde bufferlengte.
Reception callback and acknowledge	
pcmciaReceptionCallbackSet	Activeert reception callback.
pcmciaReceptionAcknowledge	Bevestigt Reception Callback.

4.3.3 Seriële interface (uart.doc)

Met deze routines kunnen we communiceren over een seriële interface. Daar wij naar verwachting in onze firmware niets gaan doen met de seriële interface, zijn de beschikbare functies hier niet opgenomen.

4.3.4 Interrupt handler(pic.doc)

Er worden een aantal routines meegeleverd waarmee wij zelf interrupts van de hardware kunnen onderscheppen en hier routines voor kunnen ontwikkelen. Aangezien het de schrijver niet geheel helder is of we hier wat mee zullen willen doen en zo ja wat, worden deze in dit document op dit moment nog niet verder beschreven.

4.3.5 Timer functies (timer.doc)

Met deze functies kan gebruik worden gemaakt van de timer in de CAM. Het is de schrijver op dit moment ook nog niet geheel duidelijk wat wij hiermee zouden gaan doen. De controlwords worden om de 2 tot 10 seconden gewisseld. Wellicht is de timer noodzakelijk om te zorgen dat wisseling van de controlwords op de juiste momenten plaatsvindt. Er zal uiteindelijk een reden zijn, waarom deze functies expliciet worden meegeleverd en gedocumenteerd.

Timer configuration	
timerInit	Initialiseert een timer.
timerOptionsGet	Vraagt de configuratie van een timer op
timerValueSet	Zet de initiële timerwaarde
timerValueGet	Verkrijgt de waarde van een timer
Timer Handling	
timerHandlerSet	Stelt het einde in van een count handler van een timer
timerHandlerGet	Verkrijgt het einde van een count handler van een timer
timerEnable	Deze functie hervat een timer
timerDisable	Deze functie stopt een timer
Extended functionality	
timerDelay	Hiermee kan een timer op een specifieke manier worden vertraagd.

4.3.6 Smartcard functies (scard.doc)

Deze verzameling functies stelt ons in staat om met smartcards te communiceren.

Inititalisatie	
scardInit	Initialiseert het gebruik van de smartcard interface
Interfacehandling	
scardOpen	Reset de kaart en wacht op de ATR (Answer to reset)
scardDelayFunctionSet	Stelt een vertragingsfunctie voor de kaart in
scardClose	Sluit de verbinding met de kaart (simuleer verwijdering van kaart)
scardByteWrite	Stuurt een karakter naar de kaart
scardByteRead	Leest een karakter in van de kaart
scardTransmitterEmpty	Controleert of er data kan worden geschreven
scardDataReady	Controleert of er data klaarstaat om te worden gelezen
Smartcard ingevoerd of verwijderd	
scardStateGet	Vraagt de status van de smartcard interface op
scardCallbackInsertSet	Activeert een callback voor invoering van een smartcard
scardCallbackExtractSet	Activeert een callback voor verwijdering van een smartcard

4.3.7 MPEG functies (mpeg.doc)

Deze belangrijke functies stellen ons in staat de MPEG stream te besturen, te filteren en te descrambelen. Bijzonder belangrijke funties voor onze firmware dus!

Inititalisatie	
mpegInit	Initialiseert het mpeg hardwareblock
mpegReset	Herinitialisatie van het mpeg hardwareblock
Filterblock	
mpegFilterSet	Hiermee kan het masker voor een filter worden ingesteld
mpegFilterGet	Hiermee kunnen het masker en de “matches” worden opgevraagd
mpegPidFiltersSet	Stelt een aantal filters in voor een PID
mpegPidFiltersGet	Hiermee kunnen de filters voor een PID worden opgevraagd
mpegPidFiltersAdd	Voegt filters toe aan een PID
mpegPidFiltersRemove	Verwijdert filters van een PID
mpegPidListFiltersSet	Hiermee kunnen filters voor meerdere PIDS worden ingesteld
mpegDataGet	Verkrijgt een gefilterd packet of sectie van een PID
mpegDataAndMaskGet	Verkrijgt een gefilterd packet of sectie van een PID en het filter waardoor de data is verkregen
Descrambling block	
mpegPidControlWordSet	Associeert een control word ID met een PID
mpegPidControlWordGet	Vraag het control word ID van een PID op
mpegPidListControlWordSet	Associeert een control word ID met meerdere PID’s
mpegControlWordSet	Kent een ID aan een control word toe
Handling Pid’s	
mpegPidStart	Start filtering/descrambling van één PID
mpegPidStop	Stopt filtering/descrambling van één PID
mpegPidSuspend	Stopt tijdelijk filtering/descrambling van één PID
mpegPidResume	Hervat filtering/descrambling van één PID
mpegPidOptionsSet	Stelt filter opties voor een PID in
mpegPidOptionsGet	Vraagt filter opties voor een PID in
mpegPidRemove	Verwijdert een PID uit het mpeg block
mpegPidListStart	Start filtering/descrambling van meerdere PID’s
mpegPidListStop	Stopt filtering/descrambling van meerdere PID’s
mpegPidListSuspend	Stopt tijdelijk filtering/descrambling van meerdere PID’s
mpegPidListResume	Hervat filtering/descrambling van meerdere PID’s
mpegPidListOptionsSet	Stelt filteropties voor meerdere PID’s in
mpegPidListOptionsRemove	Verwijdert meerdere PID’s uit het mpeg block
Globale functionaliteit	
mpegStart	Zet de hardware filtering aan
mpegStop	Zet de hardware filtering uit
mpegSuspend	Stop tijdelijk de hardware filtering
mpegResume	Hervat de hardware filtering
mpegDataSet	Voegt een packet in de mpeg stream tussen

4.4 En nu verder…

In dit hoofdstuk hebben we de bijzonderheden van de Matrix CAM’s onder de loep genomen en welke gereedschappen we voor dit type CAM beschikbaar hebben. Wat er nu nog ontbreken zijn de ontwikkelomgeving en een ontwerp van wat nu eigenlijk moeten programmeren. In het volgende hoofdstuk gaan we ons eerst op het ontwerp concentreren.

5. Technisch ontwerp

5.1 Inleiding

Hoewel de verleiding natuurlijk groot is, zullen we niet direct aan het programmeren slaan. Om tot een goed product te komen, is het verstandig eerst de doelstellingen van onze firmware te definiëren. Daarnaast is het een goed gebruik voor een te ontwikkelen systeem eerst een gestructureerd ontwerp te maken. Hierdoor wordt het programma opgebouwd uit logische routines en worden spaghettislierten voorkomen. De software wordt hierdoor ook beter onderhoudbaar.

5.2 Doelstelling

De voornaamste doelstelling van het gehele project is inzicht in de werking van firmware en de hiervoor benodigde kennis voor meer mensen toegankelijk te maken. Het is niet het doel om een emulatie te ontwikkelen waarmee zonder geldig abonnement naar zenders kan worden gekeken. Onze firmware zal gaan voorzien in het complete communicatieproces tussen de cam en de ontvanger, inclusief het aanmeldingsproces. Tevens zullen we een menustructuur ontwikkelen die eenvoudig kan worden aangepast. Wij zullen voor de codering(en) die we kunnen ondersteunen de ECM's uit de transportstream filteren en de communicatie met de smartcard waarop het officiële abonnement staat programmeren. Als het allemaal wil lukken zullen we dat in ieder geval voor de Nederlandse Canal Digitaal doen. Afhankelijk van de complexiteit hiervan kunnen we echter wellicht ook andere smartcards ondersteunen. De door de smartcard decrypted controlwords leveren we met de Sidsa API calls aan de descrambler. Het is tevens onze doelstelling EMM's te verwerken.

Daar wij de inhoud van de ECM's en de communicatie met de smartcard nog niet hebben bestudeerd, zullen wij ons in de eerste fase richten op het ontwikkelen van de firmware met de communicatie met de ontvanger en de menustructuur. Vervolgens bestuderen we de encryptie en de wijze waarop de ECM's en de EMM's aan de smartcard moeten worden aangeleverd en welke antwoorden wij hierop krijgen. In de tweede fase zullen we de functionaliteit voor de communicatie met de smartcard toevoegen.

5.3 Technisch Ontwerp

5.3.1 Schematechniek

Om onze doelstelling van gestructureerde, onderhoudbare programmatuur te realiseren gaan we eerst een ontwerp maken van onze firmware. We gaan dit doen met de voor programmeurs bekende Nassi Schneidermann PSD diagrammen (Programma Structuur Diagrammen). Zie voor een korte uitleg: http://home.tiscali.be/sectieplc.brugge/PLC/Algemeen/Analyse/06_Nassi-Schneidermann.htm.

De eerlijkheid gebied mij wel te vermelden dat de werkelijkheid achter dit document iets anders is verlopen. In versie 4 was conform deze aanpak een keurig net ontwerp gemaakt (eva_01) met een nauwkeurige scheiding in het ontwerp tussen de verschillende layers. Iedere laag had zijn eigen routines in de programmatuur, met zowel ingaande en uitgaande buffers tussen de layers van de objecten die zij bewerkten. Dit vertaalde zich echter in een code die alleen door het gebruik van de buffers al meer dan 500 Kb aan data vergde. Het is hierom dat alles opnieuw wat pragmatischer is opgezet, met als uitgangspunt het voorbeeldje van Sidsa, waarbij ieder ontvangen object afzonderlijk wordt afgehandeld. Eva_02 was geboren. Hierdoor is de onderstaande schematische voorstelling ook pas achteraf opgesteld.

Een klein probleem met deze, maar ook andere schematechnieken is dat het gebruik van globale variabelen moeilijk in beeld is te brengen. Op zich niet bijzonder want ook in programmatuur is het niet altijd even helder welke routines wijzigingen aanbrengen in de waarden van globale variabelen. En dat is wel iets waar wij een behoorlijk gebruik van gaan maken.

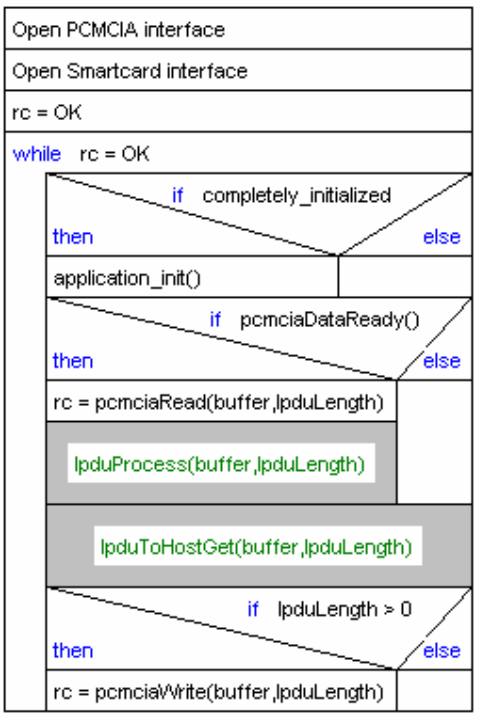
Een laatste opmerking betreft de taal die is gebruikt in zowel de schema's als in de programmatuur. De samenwerking die ik met enkele Italianen op dit vlak heb gehad, vergde dat ik het commentaar in de sourcecode in het Engels heb moeten doen. Omdat ik heb beloofd dit document ooit in het Engels te vertalen, heb ik om later extra werk te voorkomen, ook de schema's in het Engels gemaakt.

5.3.2 Structuur van de firmware

Het concept van de firmware is dat na initialisatie en aanmelding in de ontvanger continu de PCMCIA interface wordt gecontroleerd op nieuwe objecten van de ontvanger. Deze objecten worden ingelezen en indien zij gefragmenteerd worden ontvangen, geassembleerd alvorens zij worden verwerkt. Ieder object wordt dus zodra het totaal is ontvangen, direct naar de hogere layers doorgegeven en verwerkt.

Eva 02 "Emulatie voor allen"

Hoofdroutine (Main)



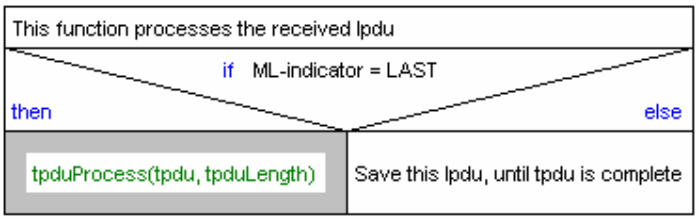
Allereerst openen we de PCMCIA interface en de Smartcard interface. In onze code gebruiken we de Smartcard interface vooralsnog alleen voor het loggen van debug-informatie.

In de application_init routine bewaken we of alle verplichte aanmeldingen hebben plaatsgevonden. We moeten ons uiteindelijk aanmelden bij de Resource Manager, Application Manager en CA Manager. De status hiervan wordt in globale variabelen bijgehouden, daar verschillende routines hier wijzigingen in zullen aanbrengen.

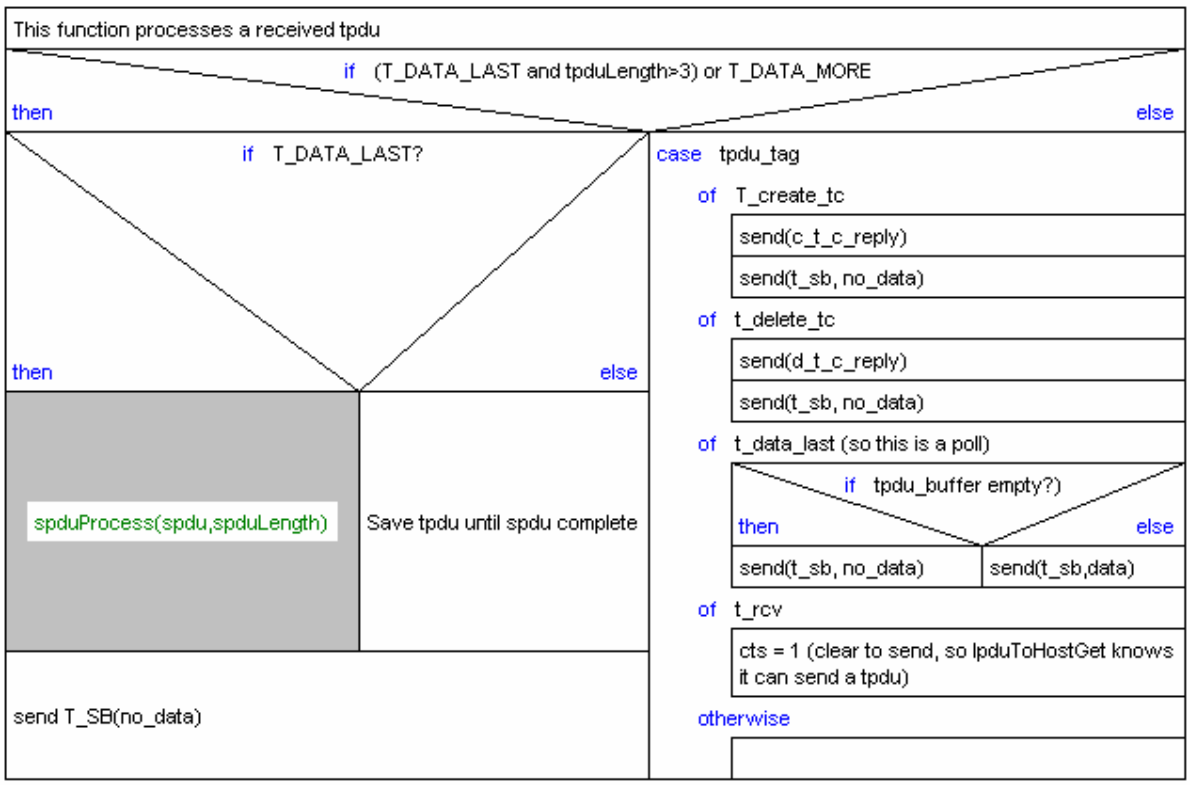
Zolang de PCMCIA interface correct werkt (status = OK), worden telkens lpdu's ontvangen en verzonden. Als er lpdu('s) te ontvangen zijn, lezen we die in met de Sidsa pcmciaRead functie en laten we deze door onze lpduProcess routine verwerken. Vanuit deze routine worden ook de hogere layers aangeroepen. Dit zal resulteren in antwoorden die in een tpdu buffer ter verzending worden geplaatst. Daar wij alleen een tpdu mogen verzenden op verzoek van of in reactie op een tpdu van de ontvanger, zal de routine lpduToHostGet controleren of er tpdu ter verzending in de buffer aanwezig is, en of deze ook mag worden verzonden. In dit geval wordt de tpdu, eventueel in meerdere lpdu's, verzonden met de Sidsa pcmciaWrite functie.

lpduProcess(buffer,lpduLength)

Deze routine verwerkt de in buffer ontvangen lpdu, met lengte lpduLength. Als de ML-indicator (more/last) gelijk is aan MORE, dan is de tpdu nog niet compleet en wordt deze bewaard om de volgende lpdu hieraan toe te voegen.
Is de ML-indicator gelijk aan LAST, dan is de tpdu compleet en zal deze worden verwerkt met de tpduProcess routine.



tpduProcess(tpdu,tpduLength)



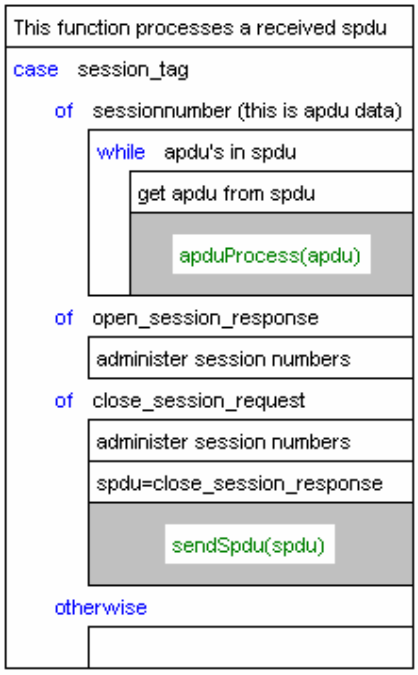
Ook al is de tpdu wel geheel ontvangen, het staat de host vrij om spdu's te segmenteren in meerdere tpdu's met de t_data_last en t_data_more tags. Bij t_data_last is de spdu compleet en kan worden behandeld. Er is een onderscheid tussen tpdu's met data voor de hogere layers en tpdu's met data waarmee de communicatie op Transport Layer niveau wordt bestuurd.

Als het een t_data_last is en de lengte is groter dan drie, of het is een t_data_more, dan hebben we zeker met data voor hogere layers te maken. We controleren dan of de spdu compleet is (t_data_last). Als hij compleet is, dan verwerken we de nu geheel ontvangen spdu met de routine spduProcess.

Is dit niet het geval dan hebben we te maken met besturingsinformatie op Transport Layer niveau. We kijken dan welke communicatie het is, en reageren hierop conform de voorschriften van EN50221. We voorzien in het verzoek tot een nieuwe transportconnection (t_create_tc), het sluiten van een bestaande transportconnection (t_delete_tc), een poll (de vraag of wij een tpdu ter verzending hebben, een lege t_data_last), en het verzoek van de ontvanger om deze dan ook te verzenden (t_rcv). De ontvanger zal ons namelijk eerst met een poll vragen of wij data hebben te verzenden en als wij hier met een t_sb(data) op antwoorden, ons een t_rcv zenden. Wij maken voor dit mechanisme gebruik van een tpdu_buffer, waarin de hogere lagen de te verzenden tpdu's klaarzetten. De routine lpduToHostGet kijkt of de globale variabele cts (clear to send) aan staat, en indien dit het geval is, zal hij een tpdu uit de buffer halen en verzenden. Onze reactie op Transport Layer niveau op de ontvangen t_rcv is dan ook de cts variable op 1 zetten, waardoor we als het ware de rem afhalen van lpduToHostGet. Na de verzending van één tpdu, eventueel in meerdere lpdu's, zet lpduToHostGet de cts variabele weer op nul. Hierdoor bereiken we dat we alleen maar tpdu's verzenden op verzoek van de ontvanger met een t_rcv.

(*Overigens bleek tijdens de tests dat de gebruikte Manhattan MX wat lichtvaardig met het EN50221 protocol omgaat. Er worden namelijk rustig t_rcv's verzonden als reactie op onze t_sb(no_data). Dus met andere worden: "Heeft u wat te verzenden?", "Nee", "OK, stuur dan maar" ☺.)*

spduProcess(spdu,spduLength)



Deze routine wordt dus aangeroepen vanuit tpduProcess als een ontvangen spdu compleet is.

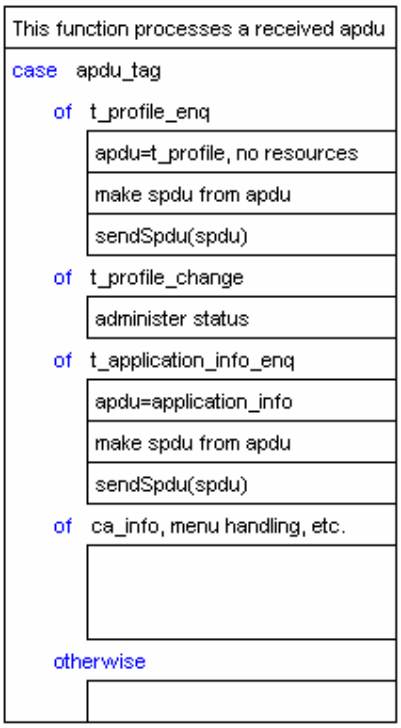
Ook hier bestaat het onderscheid tussen spdu's met data voor de hoger gelegen Application Layer en spdu's met besturingscommunicatie op Session Layer niveau.

We hoeven ons geen zorgen te maken over gefragmenteerde spdu's daar een spdu altijd één of meerdere gehele apdu's bevat. Maar we moeten er dus wel rekening mee houden dat een spdu kennelijk meerdere apdu's kan bevatten.

Een spdu met als tag Sessionnumber bevat één of meerdere apdu's. We halen stuk voor stuk uit de spdu en verwerken ze direct.

Op communicatiegebied houden we rekening met de bevestiging dat een door ons verzochte session is geopend en het verzoek van de ontvanger een bepaalde session te sluiten. De resultaten hiervan werken we bij in globale variabelen, zodat hiervan op de Application Layer gebruik gemaakt kan worden.

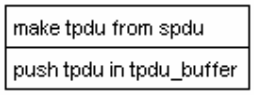
apduProcess(apdu,apduLength)



De Application Layer is natuurlijk de plaats waar de werkelijk interessante dingen plaatsvinden. In principe is het een redelijk simpele routine. Hij wordt aangeroepen met een apdu. Er wordt gekeken welke apdu het is, en hierop wordt met de juiste acties gereageerd. Vanwege de simpele structuur zijn slechts enkele apdu's ingetekend. Het principe is iedere keer hetzelfde.

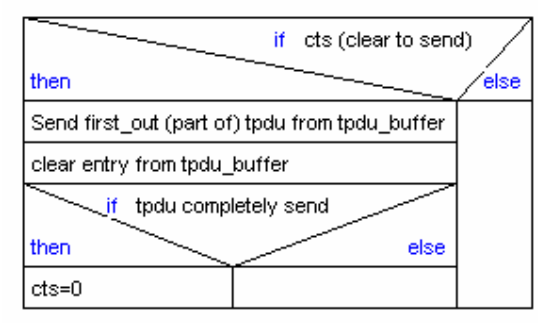
In onze huidige situatie komt het meestal neer op het beantwoorden van de apdu met de juiste apdu. In een enkel geval worden ook bepaalde gobale statusvelden bijgewerkt.

sendSpdu(spdu, spduLength)



De sendSpdu routine neemt een spdu aan en maakt hier één tpdu van. Wij splitsen dus zelf geen apdu's in t_data_last en t_data_more. Het opsplitsen van de tpdu voor verzending over de pcmcia interface vindt plaats in lpduToHostGet.
De buffer wordt bestuurd met variabelen next_in en next_out. Een nieuwe tpdu wordt in de buffer geplaatst op de index next_in, die daarna met 1 wordt verhoogd. Wanneer lpduToHostGet een tpdu uit de buffer haalt, gebeurt dit met next_out. Wanneer een tpdu uit de buffer is gehaald, wordt next_out met 1 verhoogd. Wanneer bij het verhogen van één van deze velden de maximale bufferomvang wordt bereikt, dan wordt het betreffende veld weer op 0 gezet. Zo rouleren we door de buffer.

IpduToHostGet(buffer, lpduLength)



Na ontvangst van een t_rcv object van de ontvanger wordt door tpduProcess() de cts variabele op 1 gezet. Dan mag dus een tpdu worden verzonden. De lpduToHostGet functie haalt de volgende tpdu uit de buffer (op index next_out) en kijkt of de tpdu in de pcmcia buffer past. Zo niet dan wordt de tpdu opgeknipt in meerder lpdu's waarbij de laatste de ML-indicator de waarde Last krijgt, en de voorgangers de waarde More. Als (het laatste deel van) de tpdu is verzonden, dan wordt de cts waarde weer op nul gesteld, zodat alleen maar een tpdu wordt verzonden na een t_rcv van de ontvanger. Ieder keer als deze routine wordt aangeroepen vanuit de hoofdroutine en de globale variabele cts heeft de waarde 1, wordt dus een tpdu of een deel hiervan uit de tpdu buffer verzonden. Als een tpdu geheel is verzonden, dan wordt de waarde next_out met 1 verhoogd. Of er daadwerkelijk een te verzenden tpdu zich in de buffer bevindt op de index next_out, wordt bepaald aan de hand van een aparte tabel, waarin per buffer entry de lengte van de tpdu wordt vastgelegd. Na het verzenden van een tpdu wordt deze waarde dan ook op nul gesteld.

Application_init()

Zolang niet alle vereiste aanmeldingen hebben plaatsgevonden wordt de application_init vanuit de hoofdroutine iedere keer aangeroepen. We bewaken er de juiste volgorde van de aanmeldingen. Zo zullen we pas een session openen met de Application Manager nadat de t_profile_change van de host is ontvangen. Omdat de hiervoor benodigde variabelen soms in andere routines moeten kunnen worden gewijzigd, zijn dit weer globale variabelen.

5.4 En nu verder…

Ja, nu gaat het dan toch echt gebeuren! We hebben een ontwerp van onze firmware op voldoende gedetailleerd niveau om met programmeren te kunnen beginnen. We weten de details van wat we moeten programmeren en onze gereedschapskist in bijna compleet. In het volgende hoofdstuk voegen we een ontwikkelomgeving aan onze gereedschapskist toe, checken nog één keer of we alles hebben wat we nodig hebben en dan gaan we los. Als alles lukt, dan gaan we ons na hoofdstuk 6 gaan verdiepen in de verzoeken van de ontvanger tot decodering van een zender en de werking van smartcards.

6. Programmering

6.1 Inleiding

In het vorige hoofdstuk hebben en een technisch ontwerp gemaakt voor de structuur van onze firmware. Vanzelfsprekend zullen we tijdens de programmering nog de nodige details moeten invullen. Wat we nog nodig hebben is een ontwikkelomgeving (SDK) om de door ons te schrijven C-code te compileren en te linken met de door Sidsa geleverde API-calls die wij vanuit onze code aanroepen. We zullen dus eerst de SDK bekijken. Vervolgens controleren we nog één keer of we alles wat we nodig hebben in onze gereedschapskist hebben. Dan gaan de vingers het toetsenbord op en gaan we los!

6.2 SDK (System Development Kit)

Een SDK is een ontwikkelomgeving voor een specifiek platvorm zoals MS-Windows, Unix, Linux en een specifieke soort processor. In een ontwikkelomgeving vinden we een compiler die de door ons in tekstvorm geschreven C-code controleert en omzet in een binaire objectcode. Vanuit onze code roepen we routines aan van Sidsa, die zich in een door hen geleverde library (programma bibliotheek) bevinden, de zogenaamde core libraries.

Nadat onze broncode (source) is gecompileerd, wordt deze met behulp van een linker samengevoegd (gelinked) met de door ons gebruikte functies van Sidsa. Hiervan hebben we helaas geen broncode, alleen de gecompileerde objectfiles. Het resultaat is de binary file die door de processor kan worden uitgevoerd. Deze binary wordt dus uitgevoerd, bijvoorbeeld onder Windows, maar in ons geval op de Matrix CAM door de daarin aanwezige ARM processor.

Een SDK bevat meestal ook een debugger. Hierin kan een gecompileerd en gelinked programma stap voor stap worden uitgevoerd, zodat we de waarden van variabelen en de loop van het programma kunnen controleren en eventuele fouten kunnen opsporen. Zodra we in ons geval de Sidsa routines meelinken, moeten we wel opletten dat we deze routines niet uitvoeren, omdat de hierin aangesproken hardware niet in de debugger kan worden uitgevoerd. De debugger is namelijk een ARM emulatie (in onze PC zit geen ARM processor maar bijvoorbeeld een Intel) en de hardware (MACtsp) is niet in onze pc aanwezig.

Voor de ARM processor in de Matrix waarvoor we gaan programmeren zijn verschillende (versies van) SDK's beschikbaar. Zo is bijvoorbeeld de ARK SDK 2.02, ARM SDK 2.51 en de meer recente en meest uitgebreide ADS 1.2. De links aan het einde van dit hoofdstuk geven een aantal downloadmogelijkheden.

De schrijver heeft gebruik gemaakt van ADS 1.2. De reden hiervoor was dat de enige wijze die hem mogelijk leek om debug informatie runtime te verkrijgen, het schrijven naar de smartcard is. De CAM heeft immers geen floppy disk of harddisk. Hiervoor is gebruik gemaakt van een Season interface. De standaard functie van Sidsa voor het openen van de smartcard wacht echter op de ATR (Answer To Reset) van de kaart. Omdat er geen echte kaart aanwezig was en het loggen via de Season het enige doel was, moest toch een ATR worden verzonden. Italiaanse programmeurs hebben echter een aantal functies uit de Sidsa libraries via reverse engineering weer omgezet in sourcecode en deze aangepast. Zo hebben zij een aparte variant van de smartcard functie gemaakt, scardForceOpen, die niet wacht op de ATR van de card. Het is deze functie die de schrijver heeft gebruikt. Er zijn echter verschillende formaten van libraries, die niet compatible zijn met iedere SDK. Vanwege de samenwerking met de Italianen en het door hen gebruikte ELF format, heeft de schrijver dus met ADS 1.2 gewerkt.

Hoewel er duidelijke verschillen zijn tussen de verschillende SDK's, komt het er op neer een project aan te maken, en hierin aan te geven uit welke files het project bestaat. Dit project kan vervolgens worden gecompileerd en gelinked met de build functie.

6.3 Klaar voor de start?

Alvorens te beginnen met programmeren, controleren we nu of we beschikken over alle benodigde gereedschappen en kennis.

Kennis

1. Basiskennis werking DVB uitzendingen
2. Basiskennis werking CAM / EN50221
3. Documenten EN50221 en ISO13818-1
4. Basiskennis programmeertaal C

Software

1. Een SDK Software Devlopment Kit voor ARM7 processoren
2. MM Core Libraries met de Sidsa MACtsp routines
3. Serieel communicatieprogramma zoals Realterm (voor loggen via Season interface, zie hardware)

Hardware

1. PC Met windows om op te programmeren
2. Matrix CAM (Matrix, Reloaded, Revoltutions of Reborn) of andere Sidsa based CAM
3. CAS2 interface of andere hardware om de firmware op onze Cam te laden
4. Aanbevolen: iets als een Season interface om debug info via de smartcard interface te loggen.

Dat basiskennis op het gebied van DVB uitzendingen een vereiste is, mag logisch zijn. Anders hebben we geen idee wat we aan het doen zijn. We hoeven het document EN50221 niet uit ons hoofd te kennen, maar we moeten het wel degelijk goed begrijpen. Het is ook een naslagwerk tijdens het programmeren voor de samenstellingen van de objecten. Hetzelfde geldt voor ISO13818-1.

Enige basiskennis van de programmeertaal C (of Assembly als je dat beheerst) is wel een vereiste. De schrijver is zelf absoluut geen bekwame C programmeur, maar de structuur van de taal C met zijn functies en pointers moet je wel kennen. Gelukkig beperkt ons gebruik van C zich tot het bewerken van strings, dus veel kennis is niet vereist. Ieder beginnersboek voor C zou je snel op het voor dit doel vereiste niveau moeten kunnen brengen. Zodra je weet hoe een C programma wordt opgebouwd met functies, declaraties, soorten variabelen en je kent de basis statements, kun je feitelijk al aan de slag. Voor de rest gewoon het boek ernaast houden.

Een aantal SDK's voor de ARM7 processoren is te downloaden van Internet. In de literatuurverwijzingen zijn hiervoor wat links opgenomen. Hier zijn ook de MM corelibraries te vinden.

Aangezien wij voor ARM processoren ontwikkelen met de MM corelibraries is onze firmware geschikt voor alle Sidsa gebaseerde CAM's. De schrijver heeft voornamelijk met de Matrix Revolutions gewerkt.

Na het compileren en linken van de firmware moet deze op de CAM worden geladen. Dit werkt hetzelfde als het laden van de bekende emulaties op de CAM. De schrijver heeft hiervoor een CAS2 gebruikt.

Het moeilijke met deze ontwikkeling is dat de firmware in de CAM werkt en je daar dus niet kunt debuggen. De schrijver heeft dit opgelost door in de firmware de smartcard interface te openen en hier waarden van variabelen naar toe te sturen. Via de Season in de Cam, aangesloten op de seriele poort van een notebook, kunnen met bijvoorbeeld het programma Realterm logbestanden worden gemaakt. Zo kan bijvoorbeeld het gehele communicatieproces tussen Cam en ontvanger worden gelogd en kunnen fouten worden opgespoord. Hoewel niet keihard vereist, is een dergelijke debug mogelijkheid bijna onmisbaar.

Naast alle praktische vereisten zijn er ook nog twee psychologische vereisten:
een enorm doorzettingsvermogen en oneindig veel geduld! ☺

6.4 Programmeren!

Als aan alle in de vorige paragraaf gestelde voorwaarden is voldaan, zijn we nu klaar om daadwerkelijke firmware te gaan programmeren!

Installeer een ARM SDK naar keuze en probeer eerst eens of je een “Hello world” voorbeeld kunt programmeren. Als dit lukt, weten we dat we in staat zijn een programma te schrijven, te compileren, te linken en uit te voeren.

Maak vervolgens een project aan voor de firmware. In dit project nemen we vanzelfsprekend ons bronbestand met de C-code van de firmware op. Daarnaast moeten nog de *board.c* file en de *MACDVB.32L* library worden toegevoegd aan ons project. (Het is echter ook mogelijk om met SDK utilities de routines uit de library te halen en apart mee te linken).

In de SDK moeten onder meer de volgende instellingen worden gedaan voor project, linker en compiler om tot een goed werkende code te komen.

Compiler:
- Optimalization for time

Linker:
- -FIRST hal_vectors.o -ro-base 0x04000000 -rw-base 0x1000 -NOZEROpad -map –bin

Project:
- Little Endian
- Build Release
- Target processor ARM7M

ADS 1.2: Indien wordt gewerkt met ADS 1.2 zijn nog een aantal extra instellingen nodig, zoals ARMfromELF bij de Target settings voor Postlinker, Old-style mixed-endian softfp (-fpu SoftFPA) bij ARM Assembler en Output format “Plain binary” bij ARM fromELF Linker. Bij deze laatste kunnen we dan ook de naam opgeven van de door ons gemaakte binary.

Verdere uitleg van de ARM SDK’s is hier achterwege gelaten, omdat de schrijver ook geen expert is in SDK’s. Voor hen die zich meer willen verdiepen in deze materie is echter ruimschoots voldoende informatie op internet te vinden. Hier is volstaan met het aangeven van instellingen waarmee wij tot werkende firmware kunnen komen.

En nu aan de slag!
We kunnen nu daadwerkelijk de firmware coderen, compileren, linken en in de CAM laden. Het van scratch af aan beginnen zoals de schrijver heeft gedaan kost enorm veel tijd maar is ongelooflijk leerzaam. In de MM corelibraries bevinden zich Word documenten met voorbeeldcode om te beginnen. Deze zijn echt enorm handig.

Men kan zelf op basis van deze voorbeeldcode voortborduren, een eigen technisch ontwerp maken of het door ons in hoofdstuk 5 opgestelde ontwerp programmeren.

Als houvast is in appendix 5 de broncode opgenomen van de door de schrijver voor ons doel ontwikkelde firmware. Dit geeft in ieder geval één mogelijk voorbeeld hoe firmware kan worden geschreven. Uit educatief oogmerk is enorm veel commentaar (Engels) in de code opgenomen. De meest luien onder ons zouden deze broncode natuurlijk ook gewoon kunnen overnemen en hiermee aan de slag kunnen gaan ☺.

Veel plezier!

6.5 En nu verder

We weten nu hoe we firmware kunnen schrijven, compileren, linken en op onze CAM kunnen laten werken. Hiermee kunnen we ons aanmelden in de ontvanger en hiermee communiceren. Zo zijn we in staat om een menu op het beeldscherm van onze televisie te tonen en hierin op de afstandsbediening keuzes maken, waarop we in onze firmware weer kunnen reageren.

We zullen nu gaan onderzoeken op welke wijze wij kunnen voldoen aan het verzoek van de ontvanger om bepaalde zenders te descrambelen. Dit betekent dat we ons moeten gaan verdiepen in de materie van het filteren van ECM's en de communicatie met de smartcard van onze provider. We zullen moeten weten welke vragen wij aan de smartcard moeten stellen om de voor descrambling vereiste controlwords te verkrijgen. Dit betekent kennis verkrijgen over encrypties en smartcards. Met deze kennis zullen wij vervolgens onze firmware gaan completeren.

Links:
http://www.et.fnt.hvu.nl/docenten/pkramer/ARM/ARM.htm
http://develmm.free.fr
http://opencam.altervista.org
http://www.arm.com/

7. Coderingen

8. De complete firmware

9. Smartcard operating system

Appendix 1: DVB Algemeen

(Overgenomen van : http://mccb.be.eu.org/leden/krbonne/sat-tv.belgie.html#D4)

4. DVB-S: Digital Video Broadcasting - Satellite
DVB-S is een systeem voor digitale uitzendingen van audio, video en data via satelliet. Het vormt eveneens de basis voor DVB-MS, de variant van DVB over LMDS/MVDS. Hier volgt een beschrijving over enkele belangrijkste elementen van digitale uitzendingen en DVB in het algemeen en DVB-S in het bijzonder.

4.1 Streams, kanalen en boeketten
Het eerste belangrijke verschil tussen analoge en digitale uitzendingen is dat het verband tussen een transponder en een TV-programma volledig verdwijnt. Bij analoge uitzendingen bestaat er een duidelijk één-op-één verband tussen een 'kanaal' (een transponder op een satelliet) en een TV-programma. Eén kanaal komt overeen met één programma en visa-versa. Bij digitale uitzendingen is dit helemaal niet meer van toepassing.

- Een satelliet-transponder kan één of meerdere 'transport-streams' bevatten.
- Een transport-stream kan één of meerdere TV-, radio- of data-kanalen bevatten.
- Een TV- of radio-programma bestaat uit een combinatie van een aantal audio- en/of video- en/of data-kanalen.
- Anderzijds worden verschillende programma's (mogelijk verspreid over verschillende transponders op één of meerdere satellieten) gebundeld tot één enkel boeket.

4.1.1. De transport-stream
Een 'transport-stream' is, zoals de naam het zegt, de 'transportlaag' van digitale TV. Het is een stroom van bits, uitgezonden door de satelliet, met een bepaalde snelheid, op een bepaalde frequentie en polariteit van een satelliet-transponder. Ze komt overeen met 'zoveel miljoen bits per seconde'.

Een ander verschil met analoge uitzendingen is dat een transport-stream NIET altijd de gehele transponder van de satelliet in beslag neemt. Soms kan een transport-stream slechts een deel van het spectrum van een transponder in beslag nemen, waardoor één transponder op een satelliet meerdere transport-streams kan bevatten. Soms bevindt zich op een transponder zowel een analoog TV-kanaal als een digitale transport-stream.

4.1.2 De 'PES': Packetised Elementary Stream
- Binnenin de transport-stream bevindt zich één of meerdere 'substreams'; de zogenaamde 'PES' (Packetised Elementary Stream).
- Elke PES bevat één enkele uitzending: een TV-videosignaal, een geluidskanaal, teletekstinformatie, ondertitelinginformatie, of 'pure data'.
- Elke PES kan een verschillend bitrate hebben naar gelang het type informatie dat in de sub-stream opgeslagen zit. Het spreekt voor zich dat een videosignaal meer informatie moet bevatten dan een stream die enkel ondertitelings-informatie bevat.
- Elke PES wordt geïdentificeerd aan de hand van een 'PID' (PES Id).

4.1.3 Het boeket
Een boeket is een groepering van programma's van een bepaalde aanbieder (bv. een betaal TV-aanbieder). Een boeket kan zelfs verspreid zijn over verschillende transponders van een satelliet of zelfs over meerdere satellieten op dezelfde positie.

4.2: SRs, FECs: de beschrijving van een transport-stream

Een transport-stream wordt bepaald door vier waarden:

1. Satelliet-positie (bv. 'Astra 1' op 19,2 graden oost).
2. Frequentie en polarisatie: (bv. 12,574 GHz, horizontale polarisatie)
3. SR (Symbol Rate): zie hieronder
4. FEC (Forward Error correction), zie hieronder.

4.2.1. SR: Symbol rate

Indien men een transportstream op 'technisch' niveau bekijkt, dan is dat eigenlijk een radio-draaggolf die een vast aantal keer per seconde van fase verandert. Elke faseverandering noemt men een 'symbool', en omdat men bij DVB-S gebruik maakt van QPSK-modulatie, vertegenwoordigt elk symbool 2 bits.

De symbol-rate is het aantal keer per seconde dat de transport-stream van fase verandert. Het bepaalt dus effectief de hoeveelheid informatie per seconde wordt verstuurd door de totale transport-stream.

Eén van de eigenschappen van radio-communicatie bepaalt dat de totale hoeveelheid van het radio-spectrum (de zg. bandbreedte) dat een radio-signaal heeft rechtstreeks verband houdt met het aantal maal dat het radiosignaal per second verandert.
Voor digitale uitzendingen op satelliet betekent dit dat de bandbreedte die een transport-stream nodig heeft recht evenredig is met de symbol-rate van de transportstream (want de SR is net de maat voor het aantal veranderingen per second van de draaggolf).

Veel gebruikte symbol-rates zijn 27500 of 22000 Ksymbols/s omdat de benodigde bandbreedte voor zo'n signaal net overeenkomen met wat beschikbaar is op één volledige satelliet-transponder van bepaalde satellieten. Deze SR komt tegen 2 bits per symbol overeen met 55,5 of 44 MBps aan 'pure' bitrate.

4.2.2 Fout-correctie: Solomon-Reed en FEC

Een radioverbinding is nooit perfect, en ook bij satelliet-verbindingen is het altijd mogelijk dat er fouten optreden tijdens het oversturen van een signaal.

Omdat een omroepsysteem slechts in één richting werkt (van satelliet/zender naar de ontvanger) wordt hier gebruik gemaakt van "FEC" (ofwel "Forward Error Correction"). Bij dit soort systeem wordt reeds bij het uitzenden van het radiosignaal extra informatie meegestuurd zodat -indien er fouten optreden bij het oversturen van de bits van de zender/satelliet naar de ontvanger- de ontvanger dit kan detecteren en indien mogelijk ook de fouten kan corrigeren. De error-correctie bits worden dus vooraf doorgestuurd, vandaar de naam "forward error correction".

Een transport-stream van een satelliet-verbinding wordt 'beschermd' door twee verschillende error correctie-technieken: de 'outer-coding' (meestal aangeduid met de naam 'Reed Salomon') en de 'inner-coding' (gewoon aangeduidt met 'FEC').

De eerste (Reed-Salomon) neemt een vast percentage van de transport-stream in beslag (8%) en kan niet worden gewijzigd. De tweede (FEC) is wel instelbaar en wordt aangeduid met een breuk. Een FEC van bv. '3/4' betekent dat er per 3 bits 'echte' gegevens er een 4de bit meegestuurd wordt voor error-correctie. Veel gebruikte FEC-waarden zijn 1/2, 3/4, 5/6 of 7/8.

4.2.3 Een voorbeeld: BVN
Bon, voorbeelden zijn altijd een stuk duidelijker dan saaie getallen.
Dit zijn de gegevens voor de transport-stream waarin de TV-zender 'BVN' wordt uitgezonden. (BVN = "Beste van Vlaanderen en Nederland", een zender uitgebaat door de VRT en de openbare omroepen uit Nederland)

Positie: Astra 1 (op 19,2 graden oost)
Transponder: 12,574 GHz, horizontale polarisatie
transport-stream: Symbol rate 22000 (Ksymbols/s), FEC 5/6

4.3 De PES: binnenin de transport-stream
Een transport-stream is echter maar de pure 'drager' van de binaire informatie. De eigenlijk informatie die men wenst te bekijken of te beluisteren bevindt zich 'binnenin' de transport-stream in de verschillende 'substreams' of (in het juiste vakjargon) de 'PES' (Packetised Elementry Stream) in de transport-stream.

Even herhalen wat reeds kort werd besproken in 4.1.2.

- Elke PES bevat één stroom met 'informatie'. Deze informatie kan verschillend zijn van aard. Bijvoorbeeld:
  - beeld (video)
  - geluid (mono, stereo, surround sound, ...),
  - teletext-informatie,
  - ondertitel-informatie,
  - 'pure data' (bv. internet-data).
- Elke PES wordt aangeduid via een nummer: de 'PID'.

Daarnaast bevat een transport-stream echter ook nog een hoeveelheid administratieve informatie:

- Benaming en parameters van de individuele substreams.
- Het beeldkanaal en het geluidskanaal die moeten worden gekoppeld voor het verkrijgen van een 'programma'.
- Informatie over andere transport-streams.
- 'Klok'-informatie, nodig om het geluid van een film synchroon te houden met het beeld: de PCR-stream.
- Informatie of een programma al dan niet geëncrypteerd is.
- De 'EPG' ('electronic program guide', de elektronische programmagids)

4.4 En uiteindelijk: het programma

4.4.1 Een TV-programma zoals we het nu kennen
De laatste stap die nog moet gebeuren is het opbouwen van een 'programma': datgene waar wij als kijker naar kijken. Dat verkrijgt men gewoon door verschillende substreams te combineren. Voor een TV-kanaal is dat minimaal één video-signaal samen met één geluidskanaal.

Verder moet ook worden vermeld dat er -onzichtbaar voor de gebruiker- altijd nog een 'PCR' (klok) signaal wordt meegestuurd. Dit is nodig om de verschillende datastromen synchroon met elkaar te laten lopen. Indien dit niet klopt, bv. waarbij het geluid voor of achter loopt t.o.v. het beeld, dan spreekt men over 'lip sync' problemen.

4.4.2 Een TV-programma zoals het kan zijn

Echter, één van de voordelen van digitale TV, is dat men in principe alles met alles kan combineren. Hoewel het samenvoegen van één videosignaal met één geluidssignaal de meest logische combinatie is, zijn er veel meer mogelijkheden. Enkele voorbeelden:

Een 'uitgebreid' TV-kanaal. Dit bevat naast de video-stream (het beeld) en de audio (het geluid) eveneens 'teletekst', ondertitels en de elektronische programma gids (EPG).
Soms bevat een TV-kanaal meerdere geluidskanalen: bv. verschillende talen (zoals EbS, Eurosport of Arte) of een combinatie van een 'gewoon' audio-kanaal met een 'Surround'-geluidskanaal.
Het is ook denkbaar om meerdere videokanalen samen te voegen met één enkel audiokanaal. (Bv. een sportwedstrijd bekeken vanuit verschillende camerastandpunten).
Een andere mogelijkheid: een 'onderwijs' TV-kanaal, bestaande uit beeld, geluid en een 'data'-kanaal waarlangs de slides van de cursus worden doorgestuurd.

4.4.3 Het programma: de gegevens

Al deze extra gegegevens (Audio-PID, Video-PID, enz.) bepalen samen met de gegevens van de transport-stream het TV-programma. Nemen we opnieuw het programma 'BVN' (zie 4.2.3), dan krijgen we de volgende gegevens:

De transport-stream:

- Astra 1 (op 19,2 graden oost),
- 12,574 GHz, horizontale polarisatie,
- SR 22000, FEC 5/6

De gegevens van het programma 'BVN' binnenin deze transport-stream:

- Video-PID 516
- Audio-PID 690

Appendix 2: Program Map Table (PMT) (ISO13818-1)

The Program Map Table provides the mappings between program numbers and the program elements that comprise them. A single instance of such a mapping is referred to as a "program definition." The program map table is the complete collection of all program definitions for a Transport Stream. This table shall be transmitted in packets, the PID values of which are selected by the encoder. More than one PID value may be used, if desired. The table may be segmented into one or more sections, before insertion into Transport Stream packets, with the following syntax. In each section the section number field shall be set to zero. Sections are identified by the program_number field.

Table 2-29 -- Transport Stream program map section

Syntax	No. of bits	Mnemonic
TS_program_map_section() {		
table_id	**8**	**uimsbf**
section_syntax_indicator	**1**	**bslbf**
'0'	**1**	**bslbf**
Reserved	**2**	**bslbf**
section_length	**12**	**uimsbf**
program_number	**16**	**uimsbf**
Reserved	**2**	**bslbf**
version_number	**5**	**uimsbf**
current_next_indicator	**1**	**bslbf**
section_number	**8**	**uimsbf**
last_section_number	**8**	**uimsbf**
Reserved	**3**	**bslbf**
PCR_PID	**13**	**uimsbf**
Reserved	**4**	**bslbf**
program_info_length	**12**	**uimsbf**
for (i=0; i<N; i++) {		
Descriptor()		
}		
for (i=0;i<N1;i++) {		
stream_type	**8**	**uimsbf**
Reserved	**3**	**bslbf**
elementary_PID	**13**	**uimsnf**
Reserved	**4**	**bslbf**
ES_info_length	**12**	**uimsbf**
for (i=0; i<N2; i++) {		
descriptor()		
}		
}		
CRC_32	**32**	**rpchof**
}		

table_id -- This is an 8 bit field, which in the case of a TS_program_map_section shall be always set to 0x02 as shown in table 2-27 on page 47 above.

section_syntax_indicator -- The section_syntax_indicator is a 1 bit field which shall be set to '1'.

section_length -- This is a 12 bit field, the first two bits of which shall be '00'. It specifies the number of bytes of the section starting immediately following the section_length field, and including the CRC. The value in this field shall not exceed 1021.

program_number -- program_number is a 16 bit field. It specifies the program to which the program_map_PID is applicable. One program definition shall be carried within only one TS_program_map_section. This implies that a program definition is never longer than 1016 bytes. See Informative Annex C for ways to deal with the cases when that length is not sufficient. The program_number may be used as a designation for a broadcast channel, for example. By describing the different program elements belonging to a program, data from different sources (e.g. sequential events) can be concatenated together to form a continuous set of streams using a program_number. For examples of applications refer to Annex C.

version_number -- This 5 bit field is the version number of the TS_program_map_section. The version number shall be incremented by 1 modulo 32 when a change in the information carried within the section occurs. Version number refers to the definition of a single program, and therefore to a single section. When the current_next_indicator is set to '1', then the version_number shall be that of the currently applicable TS_program_map_section. When the current_next_indicator is set to '0', then the version_number shall be that of the next applicable TS_program_map_section.

current_next_indicator -- A 1 bit field, which when set to '1' indicates that the TS_program_map_section sent is currently applicable. When the bit is set to '0', it indicates that the TS_program_map_section sent is not yet applicable and shall be the next TS_program_map_section to become valid.

section_number -- The value of this 8 bit field shall be always 0x00.

last_section_number -- The value of this 8 bit field shall be always 0x00.

PCR_PID -- This is a 13 bit field indicating the PID of the Transport Stream packets which shall contain the PCR (Program Clock Reference) fields valid for the program specified by program_number. If no PCR is associated with a program definition for private streams then this field shall take the value of 0x1FFF. Refer to the semantic definition of PCR in 2.4.3.5 on page 25 for restrictions on the choice of PCR_PID value.

program_info_length -- This is a 12 bit field, the first two bits of which shall be '00'. It specifies the number of bytes of the descriptors immediately following the program_info_length field.

stream_type -- This is an 8 bit field specifying the type of program element carried within the packets with the PID whose value is specified by the elementary_PID. The values of stream_type are specified in table 2-36 on page 64.

elementary_PID -- This is a 13 bit field specifying the PID of the Transport Stream packets which carry the associated program element.

ES_info_length -- This is a 12 bit field, the first two bits of which shall be '00'. It specifies the number of bytes of the descriptors of the associated program element immediately following the ES_info_length field.

CRC_32 -- This is a 32 bit field that contains the CRC value that gives a zero output of the registers in the decoder defined in Annex B after processing the entire Transport Stream program map section.

Appendix 3: Conditional Access Descriptor (ISO13818-1)

The conditional access descriptor is used to specify both system-wide conditional access management information such as EMMs and elementary stream-specific information such as ECMs. It may be used in both the TS_program_map_section and the program_stream_map. If any elementary stream is scrambled, a CA descriptor shall be present for the program containing that elementary stream. If any system-wide conditional access management information exists within a Transport Stream, a CA descriptor shall be present in the conditional access table.

When the CA descriptor is found in the TS_program_map_section (table_id = 0x02), the CA_PID points to packets containing program related access control information, such as ECMs. Its presence as program information indicates applicability to the entire program. In the same case, its presence as extended ES information indicates applicability to the associated program element. Provision is also made for private data.

When the CA descriptor is found in the CA_section (table_id = 0x01), the CA_PID points to packets containing system-wide and/or access control management information, such as EMMs.

The contents of the Transport Stream packets containing conditional access information are privately defined.

Table 2-52 -- Conditional access descriptor

Syntax	No. of bits	Mnemonic
CA_descriptor() {		
descriptor_tag	**8**	**uimsbf**
descriptor_length	**8**	**uimsbf**
CA_system_ID	**16**	**uimsbf**
reserved	**3**	**bslbf**
CA_PID	**13**	**uimsbf**
for (i=0; i<N; i++) {		
private_data_byte	**8**	**uimsbf**
}		
}		

CA_system_ID -- This is an 16 bit field indicating the type of CA system applicable for either the associated ECM and/or EMM streams. The coding of this is privately defined and is not specified by ITU-T | ISO/IEC.

CA_PID -- This is an 13 bit field indicating the PID of the Transport Stream packets which shall contain either ECM or EMM information for the CA systems as specified with the associated CA_system_ID. The contents (ECM or EMM) of the packets indicated by the CA_PID is determined from the context in which the CA_PID is found, i.e. a TS_program_map_section or the CA table in the Transport Stream, or the stream_id field in the Program Stream.

Appendix 4: Specificaties Matrix

1.	**Multimedia engine**: 144 pins - SIDSA MACTSP 0234 56524A - R2G6632 SIDSA's MACtsp multimedia engine is een enkele chip, gebaseerd op de snelle ARM7TDMITM processor die nog slechts enkele externe componenten nodig heeft: SRAM, FLASH en Smart Card Driver. Verschillende Conditional Access systemen kunnen worden geïmplementeerd op dezelfde engine door de software development tools en referentie ontwerpen van SIDSA.
2.	**Flash memory**: 48 pins - M29W160DB - 16 Mbit (2Mb x8 or 1Mb x16, Boot Block) The M29W160D is een 16 Mbit (2Mb x8 or 1Mb x16) non-volatile memory dat kan worden gelezen, gewist en geprogrammeerd. Bij het opstarten wordt het memory in default Read modus gezet waarbij het op dezelfde manier als ROM of EPROM kan worden gelezen. Het geheugen is ingedeeld in blokken die onafhankelijk van elkaar kunnen worden gewist zodat het mogelijk is bepaalde gegevens te behouden, terwijl andere gegevens worden gewist. Elk blok kan worden beschermd om te voorkomen dat de gegevens per ongeluk worden gewist. Programma en Erase commando's worden geschreven naar de Command interface van het geheugen. Een on-chip Program/Erase Controller versimpelt het proces van programmering of wissen van het geheugen door alle speciale acties te verzorgen die zijn benodigd om het geheugen te bewerken. Het einde van een programmeer of wis actie kan worden gedetecteerd en errors kunnen worden geïdentificeerd. De instructieset om het geheugen te beheren is consistent met JEDEC standaarden. De geheugenblokken zijn assymetrisch ingedeeld. De eerste of laatste 64 Kbytes zijn ingedeeld in 4 additionele blokken. Het 16 Kbyte Boot Block kan worden gebruik voor een kleine initialisatiecode om de microprocessor te starten. De twee 8 Kbyte Parameter Blocks kunnen worden gebruikt voor opslag van parameters. De overige 32 Kbytes is een klein hoofdblok waar de applicatie kan worden opgeslagen. Chip Enable, Output Enable en Write Enable signalen controleren de bus operation van het geheugen. Hierdoor kan eenvoudig verbinding worden gemaakt met de meeste microprocessoren, vaak zonder additionele programmeerlogica. Het geheugen wordt aangeboden in SO44, TSOP48 (12 x 20 mm) en TFBGA48 (0,8 mm picth) pakketten. Het geheugen wordt geleverd met alle gewiste bits op '1' gezet.
3.	**Xilinx**: 44 pins XC9536XL De XC9536XL is een 3.3V CPLD (Complex Programmable Logical device), Gericht op applicaties met een hoge performance en een lag voltage op het gebied van communciatie en computer systemen. Hij bestaat uit twee 54V18 Function Blocks, waardoor 800 usable gates worden geboden "with propagation delays of 5 ns." (lekker duidelijk! ☺)
4.	**32 pins - B002C - 8174S** ???
5.	**CMOS SRAM**: 44 pins - Alliance - AS7C34098-15TC 44 pins - Alliance - AS7C34098-15TC is a 3.3V 256K x 16 CMOS SRAM, access time 15ns made by Alliance Semiconductor Corporation. The AS7C34098 is a high-performance CMOS 4,194,304-bit Static Random Access Memory (SRAM) devices organized as 262,144 words × 16 bits. They are designed for memory applications where fast data access, low power, and simple interfacing are desired.

Appendix 5: Sourcecode firmware

Onderstaand de door de schrijver ontwikkelde source voor firmware voor de Sidsa based CAM's, zoals o.a. de Matrix in al zijn generaties. De code is getest op een Matrix Revolutions. Op zichzelf doet de code nog niets meer dan een menu op het beeldscherm zetten met submenus. Het laat echter wel zien hoe de volledige communicatie met de ontvanger, inclusief de aanmeldingen kan verlopen. Door het overhevelen van de tekst vanuit de programeditor naar Word, zijn de tabs op sommige plekken wat versprongen.

```
// EVA Emulatie voor allen.
// Versie 2.0
// Hermanator
// 20/03/2005
// ================================================================================================
#include "hal.h"
#include "error.h"
#include "pcmcia.h"
#include "scard.h"
#include <string.h>

#define DEBUG                                       // When debug, open and write debug info on smartcard

#define PCMCIAs_BUF_LEN                      200    // Bufferlength pcmcia interface
#define TPDU_LEN                            2048    // Maximum length tpdu packet
#define MAX_TPDU                              20    // Maximum size of tpdu-sendbuffer
#define SPDU_LEN                            2048    // Maximum length spdu packet
#define APDU_LEN                            2048    // Maximum length adpu packet

// Link layer tags
#define MORE                                0x80
#define LAST                                0x00

// Transport layer tags
#define T_SB                                0x80
#define T_RCV                               0x81
#define T_CREATE_TC                         0x82
#define T_C_T_C_REPLY                       0x83
#define T_DELETE_T_C                        0x84
#define T_D_T_C_REPLY                       0x85
#define T_REQUEST_TC                        0x86
#define T_NEW_TC                            0x87
#define T_T_C_ERROR                         0x88
#define T_DATA_LAST                         0xA0
#define T_DATA_MORE                         0xA1

#define DATA                                0x80
#define NO_DATA                             0x00

// Session layer tags
#define OPEN_SESSION_REQUEST                0x91
#define OPEN_SESSION_RESPONSE               0x92
#define CREATE_SESSION                      0x93
#define CREATE_SESSION_RESPONSE             0x94
#define CLOSE_SESSION_REQUEST               0x95
#define CLOSE_SESSION_RESPONSE              0x96
#define SESSION_NUMBER                      0x90

// Application layer tags
static unsigned char t_profile_enq[3]              ={0x9f,0x80,0x10};
static unsigned char t_profile[3]                  ={0x9f,0x80,0x11};
static unsigned char t_profile_change[3]           ={0x9f,0x80,0x12};
static unsigned char t_application_info_enq[3]     ={0x9f,0x80,0x20};
static unsigned char t_application_info[3]         ={0x9f,0x80,0x21};
static unsigned char t_enter_menu[3]               ={0x9f,0x80,0x22};
static unsigned char t_ca_info_enq[3]              ={0x9f,0x80,0x30};
static unsigned char t_ca_info[3]                  ={0x9f,0x80,0x31};
static unsigned char t_ca_pmt[3]                   ={0x9f,0x80,0x32};
static unsigned char t_ca_pmt_reply[3]             ={0x9f,0x80,0x33};
static unsigned char t_close_mmi[3]                ={0x9f,0x88,0x00};
static unsigned char t_display_control[3]          ={0x9f,0x88,0x01};
static unsigned char t_display_reply[3]            ={0x9f,0x88,0x02};
```

```
static unsigned char t_text_last[3]               ={0x9f,0x88,0x03};
static unsigned char t_text_more[3]               ={0x9f,0x88,0x04};
static unsigned char t_enq[3]                     ={0x9f,0x88,0x07};
static unsigned char t_answ[3]                    ={0x9f,0x88,0x08};
static unsigned char t_menu_last[3]               ={0x9f,0x88,0x09};
static unsigned char t_menu_more[3]               ={0x9f,0x88,0x0a};
static unsigned char t_menu_answ[3]               ={0x9f,0x88,0x0b};
static unsigned char t_list_last[3]               ={0x9f,0x88,0x0c};
static unsigned char t_list_more[3]               ={0x9f,0x88,0x0d};

// Resources in host
unsigned char resource[4][4] =       {0x00,0x01,0x00,0x41,         // Resource Manager RM
                                      0x00,0x02,0x00,0x41,         // Application Manager AM
                                      0x00,0x03,0x00,0x41,         // CA Manager CA
                                      0x00,0x40,0x00,0x41};        // MMI
unsigned char session[4][2];                      // Administration of sessionnumbers
unsigned char session_state[4];                   // Session states: see defines closed,requested and open
unsigned char session_request_tc_id;              // Transportconnection to use requesting session

#define RM     0
#define AM     1
#define CA     2
#define MMI    3

#define CLOSED         0
#define REQUESTED      1
#define OPEN           2


// Menu definition
#define MAX_MENU_OPTIONS      8

unsigned char toplevel[7]={"Eva_02"};                              // Name of emulation
unsigned char top_subheader[30]={"Open source Matrix firmware"};   // Subheader of main menu
unsigned char footer[20]={"Make your choice"};                     // Standard footer
unsigned char menu_pos[2];                                         // Current position in menu
unsigned char menu_option[MAX_MENU_OPTIONS][MAX_MENU_OPTIONS][20] =
{       {"Card info","Settings","Edit keys","Serial update","Card update","","",""},
        {"No choices","","","","","","","",""},
        {"No access message","FRC","CryptW AU","Nagra AU","Emu key","Default keys","",""},
        {"Viaccess 1","Viaccess 2","Viaccess 3","Seca 1","Seca 2","Irdeto","Nagra","Biss"},
        {"No choices","","","","","","","",""},
        {"No choices","","","","","","","",""},
        {"","","","","","","","",""},
        {"","","","","","","","",""},
};
#define MENU_MAIN             0
#define MENU_CARD_INFO        1
#define MENU_SETTINGS         2
#define MENU_EDIT_KEYS        3
#define MENU_SERIAL_UPDATE    4
#define MENU_CARD_UPDATE      5


// In menu_option[][][] we find the description of the menu options and submenu options.
// So we have a menu of maximum two levels. If a choice is made, the menuheader is
// allways toplevel, the subheader is the choice made in the higer level. The footer
// is the standard footer. In our program the current menu position is saved in menu_pos.
// If menu_pos = "60" we are within the sixth choice of the main menu.
// If menu_pos = "62" we are within the second choice of the sixth main menu choice.
// If menu_pos = "00" we are in the main menu
// By putting menu options in an array we can dynamically change them if required

// The caid's we can handle, for the time being, fakes
static unsigned char caid[3][2] =    {0x00,0x01,
                                      0x00,0x02,
                                      0x00,0x03};
static unsigned int nr_caids  = 3;

//Smartcard defines
#define SCARDs_ATR_SIZE       (sizeof (scardS_atrValid))
#define SCARDs_NUM_CARD       0
#define SCARDs_FIRST_START    0
#define SCARDs_FIRST_END      7
#define SCARDs_SECOND_START   10
#define SCARDs_SECOND_END     16
```

```
#define SCARDs_ATR_VALID      {0x3B, 0xF7, 0x11, 0x00, 0x01, 0x40, 0x96,0x70, 0x70, 0x07, 0x0E, 0x6C,
0xB6, 0xD6, 0x90, 0x00}
#define SCARDs_OPERATION_SEND {0xC1, 0x0E, 0x00, 0x00, 0x08}
#define SCARDs_OPERATION_SIZE 11

static char scardS_atrValid[] = SCARDs_ATR_VALID;

// Buffer definitions for tpdu-send buffer. Since we are only allowed to send when requested by the host,
// we use a cts flag to see if we can send or not. When cts=1 then we can send a tpdu.
// We make a seperate table for the tc_id of a tpdu, whichs saves us the hassle of extracting it from
// our own tpdu's when building lpdu's. It is a rotating buffer. We start filling at 0 until we reach
// the end of the buffer. Then we start at 0 again. T_next_in indicates the place in the buffer to
// put the first tpdu to be sent. T_next_out indicates the first tpdu to be extracted and sent.
// When t_next_out equals t_next_in, the buffer is empty. We check if there is an entry in the buffer
// or not with the t_buffer_len. If this is zero, the entry is empty, allthough old values can still reside.
// Buffersizes can be changed to resolve buffer issues with MAX_TPDU and TPDU_LEN.

unsigned char t_buffer[MAX_TPDU][TPDU_LEN];         // Buffer for tpdu's to be sent
long t_buffer_len[MAX_TPDU];                        // Lengths of buffer entries
unsigned char t_tc_id[MAX_TPDU];                    // Transportconnection of buffer entries
unsigned int t_next_in=0;                           // Pointer for next entry in buffer
unsigned int t_next_out=0;                          // Pointer for next extraction of buffer
unsigned int cts=0;                                 // clear to send, 0 = no, 1 = yes

unsigned char cLpdu[PCMCIAs_BUF_LEN];               // For transportcommunication bypassing buffer
unsigned char cLpduLength;                          // Length of cLpdu;

// Communication functions
int lpduProcess (unsigned char buffer[PCMCIAs_BUF_LEN],unsigned char lpduLength);
int lpduToHostGet(unsigned char* buffer,unsigned char* lpduLength);
int tpduProcess (unsigned char* tpdu, long tpduLength);
int spduProcess (unsigned char* spdu, long spduLength, unsigned char tc_id);
int apduProcess (unsigned char* apdu, long apduLength, unsigned char tc_id);
int push_tpdu(unsigned char* tpdu,long tpduLength, unsigned char tc_id);
int send_spdu (unsigned char* spdu, long spduLength, unsigned char tc_id);
int request_session_resource(unsigned char resource[]);
int application_init(void);

// Menu functions
int send_menu (unsigned char menu, unsigned char* apdu, long apduLength, unsigned char tc_id);

//General purpose functions
int equal(unsigned char string1[], unsigned int van, unsigned int tm, unsigned char string2[]);
int strlength(unsigned char string[]);
int calcLengthfield(long length, unsigned char lengthfield[]);

#ifdef DEBUG
int scardLog (unsigned char logmsg[],long logmsgLength);
int hexconvert (unsigned char* string, unsigned char Length, unsigned char* hextstring);
unsigned char logmsg[512];
long logmsgLength;
unsigned char hexstring[512];
#endif

// Main program
// ================================================================================================================
int main( int i_argc,char* i_argv[]) {

        unsigned long rc;
        unsigned char buffer[PCMCIAs_BUF_LEN];
        unsigned char bufferLength = PCMCIAs_BUF_LEN;
        unsigned char lpduLength;

        rc = pcmciaOpen( &bufferLength);
Open pcmcia interface
        rc = OK;
                                                    // Seems to be necessary...
        if( (rc == OK) || (rc == PCMCIA_ERROR_LEN_NOT_ACCEPTED) ) {
#ifdef DEBUG
        //Inizialize Smart Card
        scardOptions_t options;                     // Structure for opening scard interface
        char atr[SCARDs_ATR_SIZE];                  // Variable for ATR
        scardState_t cardState;
        scardInit (SCARDs_NUM_CARD);
        options.ATR = atr;
        options.baudRate  = SCARD_BAUD_9600;        // 9600 bps
```

```
	options.ATRLength =1;                                   //SCARDs_ATR_SIZE;forced for debug
	scardStateGet(0, &cardState);
//	if(cardState == SCARD_INSERTED);
//		scardOpen (SCARDs_NUM_CARD, &options);
	scardForceOpen (SCARDs_NUM_CARD, &options);
	logmsgLength=6;                                         // Indicate incoming message
	scardLog(toplevel, logmsgLength);                       // Log it on smartcard
#endif
	// Main program reading and writing pcmcia
	rc = OK;                                                // Seems to be necessary...
	while(rc == OK) {
		if (session_state[CA] != OPEN) application_init();  // Check if all necessary sessions are opened
			if(pcmciaDataReady() == TRUE) {             // Check if a lpdu has been received from host
				lpduLength = bufferLength;
				rc = pcmciaRead(buffer, &lpduLength);               // Read received lpdu
				if( (rc == OK) && (lpduLength != 0) && (lpduLength <PCMCIAs_BUF_LEN )) {
#ifdef DEBUG
					logmsg[0]='>'; logmsg[1]=' ';               // Indicate incoming message
					hexconvert(buffer, lpduLength, hexstring);
                                                                    // Put buffer in hexformat for notepad
					memcpy((&logmsg[2]),hexstring,lpduLength*2); // put hexstring in logmsg;
					logmsgLength=lpduLength*2+2;                // Total length of logmsg;
					scardLog(logmsg, logmsgLength);             // Log it on smartcard
#endif
					lpduProcess(buffer, lpduLength); // Extract tpdu from lpdu and process it
				}
			}
			lpduLength = bufferLength;
			rc = lpduToHostGet(buffer, &lpduLength);    // Check if a lpdu has to be sent to host
			if( (rc == OK) && (lpduLength != 0) ) {
				rc = pcmciaWrite( buffer, lpduLength);      // Throw it on the pcmcia interface
#ifdef DEBUG
				logmsg[0]='<'; logmsg[1]=' ';               // Indicate incoming message
				hexconvert(buffer, lpduLength, hexstring);  // Put buffer in hexformat for notepad
				memcpy((&logmsg[2]),hexstring,lpduLength*2); // put hexstring in logmsg;
				logmsgLength=lpduLength*2+2;                // Total length of logmsg;
				scardLog(logmsg, logmsgLength);             // Log it on smartcard
#endif
			}
			// Perform other operations (MPEG filters processing, ...)
		}
	}
	return OK;
}

// Link layer functions
// ============================================================================================================================
int lpduToHostGet(unsigned char* buffer,unsigned char* lpduLength) {

	static offset=0;                                    // Pointer to buffer tpdu to start sending data
	unsigned char databytes;                            // The number of databytes to be put in lpdu
	lpduLength[0]=0;
	if (cts) {
		if (cLpduLength>0) {                        // Transport communication transport layer first!
			memcpy(buffer,cLpdu,cLpduLength);   // return the cLpdu
			lpduLength[0]=cLpduLength;          // Length of cLpdu
			cLpduLength=0;                      // clear cLpdu buffer
			cts=0;                              // we will have to wait for poll from host again
			return OK;
		}
	}

	if (cts) {                                          // Allowed to send data at the moment?
		databytes=t_buffer_len[t_next_out]-offset;  // Number of bytes from tpdu to be sent
		buffer[0]=t_tc_id[t_next_out];              // Fill transport connection from buffer
		if (databytes>PCMCIAs_BUF_LEN-2) {          // More data than fits in buffer
				buffer[1]=MORE;             // ML-indicator = more
				databytes=PCMCIAs_BUF_LEN-2; // Restrict databytes to maximum
				memcpy((&buffer[2]),t_buffer[t_next_out],databytes);
                                                            // Copy (part of) tpdu in lpdu buffer
				offset+=databytes;          // Move offset to next part of tpdu
		}
		else { buffer[1]=LAST;                      // ML-indicator = last
				memcpy((&buffer[2]),t_buffer[t_next_out],databytes);
                                                            // Copy (part of) tpdu in lpdu buffer
				offset=0;                   // Reset to start of tpdu
```

```
                        t_buffer_len[t_next_out]=0;   // Clear buffer entry
                        t_next_out++;                 // Set buffer pointer to next tpdu
                        if(t_next_out>=MAX_TPDU) t_next_out=0; // If eof buffer, restart at beginning
                        cts=0;                        // Not allowed to send till next T_RCV
                }
                lpduLength[0]=databytes+2;            // Length lpdu = length tpdu + header lpdu
                return OK;
        }
        else {
                return FALSE;                         // Not allowed to send data at the moment
        }
        return OK;
}

int lpduProcess (unsigned char buffer[PCMCIAs_BUF_LEN],unsigned char lpduLength) {

        static unsigned char tpdu[TPDU_LEN];          // buffer for received tpdu
        static long tpduLength=0;                     // Total length of received tpdu

        memcpy((&tpdu+tpduLength),(&buffer[2]),lpduLength-2); // add tpdu data from lpdu to receive tpdu buffer
        tpduLength+=(lpduLength-2);                   // Add received tpdu length to length tpdu
        if (buffer[1] == LAST) {                      // then tpdu is complete, if not wait for the rest
                tpduProcess(tpdu,tpduLength);         // process received tpdu
                tpduLength=0;                         // reset length to zero for next tpdu
        }
        return OK;
}

// Transport layer functions
// ================================================================================================================
int tpduProcess (unsigned char* tpdu, long tpduLength) {

        static unsigned char spdu[SPDU_LEN];                    // buffer for received spdu
        long spduLength=0;                                      // Total length of received spdu
        unsigned int lengthbytes;                               // Number of length bytes in received tpdu
        long len_spdu;                                          // Length of the received spdu inside tpdu
        unsigned char tc_id;                                    // transport connection of received tpdu
        if (tpdu[1]<=127) lengthbytes=1;                        // one length byte
        else lengthbytes=1+(tpdu[1] - 127);                     // more then one lengthbyte
        tc_id=tpdu[0+lengthbytes+1];                            // Get the transportconnection
        len_spdu=tpduLength-1-lengthbytes-1;                    // length spdu = tpdulength minus tpdu-header

        if ((tpdu[0]==T_DATA_LAST && tpduLength>3) || tpdu[0]==T_DATA_MORE) { // spdu data received
                memcpy((&spdu+spduLength),(&tpdu[1+lengthbytes+1]),len_spdu);
                                                                // move spdu data to spdu buffer at offset
                spduLength+=len_spdu;                           // add nr of databytes to total length spdu
                if (tpdu[0]==T_DATA_LAST) {                     // spdu complete
                        spduProcess(spdu, spduLength, tc_id);   // process spdu
                        spduLength=0;                           // clear spdu buffer after processing it
                }
                cLpdu[0]=tc_id; cLpdu[1]=LAST;                  // cLpdu header
                cLpdu[2]=T_SB; cLpdu[3]=0x02; cLpdu[4]=tc_id;   // T_sb, acknowledge
                cLpdu[5]=NO_DATA;                               // with no_data
                cLpduLength=0x06;                               // which adds up to 4 bytes
                cts=1;                                          // Clear to send
        }
        else {                                                  // Transportcontrol data received
                switch(tpdu[0]) {
                case T_CREATE_TC:                               // Send c_t_c_reply immediately in cLpdu
                        cLpdu[0]=tc_id; cLpdu[1]=LAST;                          // lpdu header
                        cLpdu[2]=T_C_T_C_REPLY; cLpdu[3]=0x01; cLpdu[4]=tc_id;  // c_t_c_reply
                        cLpdu[5]=T_SB; cLpdu[6]=0x02; cLpdu[7]=tc_id;           // T_sb
                        if (session_state[RM] == CLOSED)        // Do we have to send a session request to RM?
                                cLpdu[8]=DATA;                  // DATA if we have to connect to RM
                        else    cLpdu[8]=NO_DATA;               // else no DATA (second or more tc_id)
                        cLpduLength=0x09;                       // which adds up to 9 bytes
                        cts=1;                                  // Clear to send
                        session_request_tc_id=tc_id;            // For requesting sessions later on
                        break;
                case T_DELETE_T_C:                              // Send c_t_c_reply immediately in cLpdu
                        cLpdu[0]=tc_id; cLpdu[1]=LAST;                          // lpdu header
                        cLpdu[2]=T_D_T_C_REPLY;cLpdu[3]=0x01;cLpdu[4]=tc_id;    // D_t_c_reply
                        cLpdu[5]=T_SB;cLpdu[6]=0x02;cLpdu[7]=tc_id;             // T_sb
                        cLpdu[8]=NO_DATA;                                       // with no_data
                        cLpduLength=0x09;                                       // which adds up to 9 bytes
                        cts=1;                                                  // Clear to send
```

```
                break;
        case T_DATA_LAST:                                       // Poll: host: "anything to say?"
                cLpdu[0]=tc_id; cLpdu[1]=LAST;                  // lpdu header
                cLpdu[2]=T_SB;cLpdu[3]=0x02;cLpdu[4]=tc_id;     // First part of T_sb
                if (t_buffer_len[t_next_out]>0) cLpdu[5]=DATA;
                                                // data in buffer, Cam: "I've got something for you"
                else    cLpdu[5]=NO_DATA;       // no data in buffer, Cam: "sorry, nothing to say"
                cLpduLength=0x06;
                cts=1;                          // Clear to send
                break;
        case T_RCV:                             // T_rcv, host says "give it to me baby!"
                if (t_buffer_len[t_next_out] > 0)   // Check if we really have something to send???
                        cts=1;                  // If we do, take the break off lpdutohostget
                else {  cLpdu[0]=tc_id;         // if not, send t_sb with no data
                        cLpdu[1]=LAST;cLpdu[2]=T_SB;cLpdu[3]=0x02;
                                                // I have done this because my Manhattan sometimes
                        cLpdu[4]=tc_id;cLpdu[5]=NO_DATA;
                                                // replies to a T_sb with no_data with a t_rcv (???)
                        cLpduLength=0x06;       // and this way it works.
                        cts=1;
                }
                break;
        }
    }
    return OK;
}

int push_tpdu(unsigned char* tpdu,long tpduLength, unsigned char tc_id) {
    memcpy(&t_buffer[t_next_in],tpdu,tpduLength);       // Push tpdu in buffer
    t_buffer_len[t_next_in]=tpduLength;                 // Register length tpdu in buffer
    t_tc_id[t_next_in]=tc_id;                           // Register transportconnection in buffer
    t_next_in++;                                        // Pointer to next entrypoint in buffer
    if (t_next_in>=MAX_TPDU) t_next_in=0;               // If eof buffer, than restart at beginning
    return OK;
}

// Session layer functions
// ========================================================================================================================
int spduProcess (unsigned char* spdu, long spduLength, unsigned char tc_id) {

    static unsigned char apdu[APDU_LEN];                // buffer for received apdu
    long apduLength=0;                                  // Total length of received apdu('s)
    unsigned int lengthbytes=0;                         // Number of length bytes in received spdu
    long datalength=0;                                  // Number of databytes in received spdu
    long begin_apdu;                                    // Pointer to beginning of next apdu in spdu
    long end_apdu;                                      // Pointer to end of next apdu in spdu
    unsigned int i=0;                                   // counter for/next
    unsigned char session_status;                       // replyfield in session close
    unsigned char sessionnr[2];                         // Sessionnumber
    switch (spdu[0]) {
        case SESSION_NUMBER:                            // apdu data received! extract and process it
            begin_apdu=4;                               // start of first apdu in spdu
            while (begin_apdu<spduLength) {             // while spdu contains more apdu's
                if (spdu[begin_apdu+3]<=127) {          // how many length bytes do we have?
                    lengthbytes=1;                      // one length byte
                    datalength=spdu[begin_apdu+3];      // then length in second byte
                }
                else {                                  // more then one lengthbyte present
                    lengthbytes=1+(spdu[begin_apdu+3] - 127);
                                                        // Calculate number of lengthbytes and datalength
                    for (i=lengthbytes;i>0;i--)
                        datalength+=spdu[begin_apdu+3+i]*((lengthbytes-i)*256);
                }
                end_apdu=begin_apdu+3+datalength+lengthbytes-1;     // determine end of apdu
                memcpy(&apdu,spdu,4);                               // copy session header in apdu
                memcpy(&apdu[4],(&spdu[begin_apdu]),end_apdu-begin_apdu+1);// copy apdu from spdu
                apduLength=4+end_apdu-begin_apdu+1;                 // total length of packet
                apduProcess(apdu,apduLength,tc_id);                 // process received spdu
                begin_apdu=end_apdu+1;                              // pointer to next apdu in spdu
            }
            break;

        case OPEN_SESSION_RESPONSE:                     // Our requested session is now open if status=0x00
            for (i=0;i<4;i++) {                         // Check all 4 resources
                if (equal(spdu,3,6,resource[i])) {      // If it's the one,
                    if (spdu[2] == 0x00)                // If the session is succesfully opened:
```

```
                                        memcpy(&session[i],&spdu[7],2);
                                                // put the sessionnumber in it's entry in the table
                                else    memset(&session[i],0x00,2);   // if not, blank out sessionnumber
                        }
                }
                if (equal(spdu,3,6,resource[MMI]) && spdu[3] == 0x00) {     // When session = MMI
                        spdu[0]=0x90;                                       // Session data
                        spdu[1]=0x02;                                       // two bytes of data
                        memcpy(&spdu[2],session[MMI],2);                    // Put sessionnr MMI in
                        memcpy(&spdu[4],&t_display_control,3);              // Insert display control tag
                        spdu[7]=0x02;                                       // Lengthfield apdu = 2 bytes
                        spdu[8]=0x01;                                       // display_cmd = set_mmi_mode
                        spdu[9]=0x01;                                       // high_level_mmi
                        spduLength=10;                                      // a total of 10 bytes
                        send_spdu(spdu,spduLength,tc_id);                   // Throw it out
                }
                break;

        case CLOSE_SESSION_REQUEST:                                         // Host wants to close session
                for (i=0;i<4;i++) {                                         // Check all 4 resources
                        if (equal(spdu,2,3,session[i])) {                   // find resource of session to close
                                memset(&session[i],0x00,2);                 // blank out sessionnumber
                                session_state[i]=CLOSED;                    // Remember session is closed
                                session_status=0x00;                        // Acknowledge closure
                        }
                }
                memcpy(&sessionnr,&spdu[2],2);                              // Get sessionnumber from received spdu
                spdu[0]=CLOSE_SESSION_RESPONSE; spdu[1]=0x03;               // Assembly response
                spdu[2]=0x00; memcpy(&spdu[3],sessionnr,2);
                spduLength = 0x05;                                          // Length of spdu
                send_spdu(spdu,spduLength,tc_id);                           // Send acknowledgement
                break;
        }
        return OK;
}

int send_spdu (unsigned char* spdu, long spduLength, unsigned char tc_id) {

        unsigned char tpdu[TPDU_LEN];                                       // tpdu to be sent
        long tpduLength;                                                    // length of tpdu to be sent
        int lengthbytes;                                                    // number of lengthbytes
        unsigned char lengthfield[3];                                       // the lengthfield for the tpdu

        tpdu[0]=T_DATA_LAST;                                                // We allways make 1 tpdu, never split it up
        lengthbytes=calcLengthfield(spduLength+1,lengthfield);              // Calculate the lengthfield (add 1 for tc_id)
        memcpy(&tpdu[1],lengthfield,lengthbytes);                           // copy in lengthfield
        tpdu[1+lengthbytes]=tc_id;                                          // transportconnection
        memcpy(&tpdu[2+lengthbytes],spdu,spduLength);                       // cpy spdu in tpdu
        tpdu[spduLength+2+lengthbytes]=T_SB;
        tpdu[spduLength+3+lengthbytes]=0x02;
        tpdu[spduLength+4+lengthbytes]=tc_id;
        tpdu[spduLength+5+lengthbytes]=NO_DATA;                             // Add T_sb no data
        tpduLength=spduLength+6+lengthbytes;                                // Total length tpdu
        push_tpdu(tpdu,tpduLength,tc_id);                                   // push tpdu in tpdu-sendbuffer
        return OK;
}

// Application layer functions
// ===========================================================================================================
int apduProcess (unsigned char* apdu, long apduLength, unsigned char tc_id) {

        unsigned int i;                                         // for/next counter
        unsigned int lengthbytes;                               // Field for number of lengthbytes in apdu
        long len_apdu;                                          // Value of lengthfield
        static unsigned int menu=0;                             // The menu to be displayed, remember current menu

        if (apdu[7]<=127) {                                     // Calculatie lengthfield en length apdu
                        lengthbytes=1;                          // one length byte
                        len_apdu=apdu[7];                       // Value of lengthfield within apdu
        }
        else {  lengthbytes=1+(apdu[7] - 127);                  // more then one lengthbyte
                        len_apdu=0;
                        for (i=lengthbytes; i>0; i--)
                                len_apdu+=apdu[7+i]*((lengthbytes-i)*256);  // Value of lengthfield within apdu
        }
        if (equal(apdu,4,6,t_profile_enq)) {                    // Resource Manager asks for profile
```

```
                memcpy(&apdu[4],&t_profile,3);                      // Let's give it! (retain same session header)
                apdu[7]=0x00;                                       // We have no resources to share
                apduLength=0x08;                                    // adds to a total of 8 bytes packetlength
                send_spdu(apdu,apduLength,tc_id);                   // If fact we already builded a real spdu
        }
        if (equal(apdu,4,6,t_profile_change)) {                     // Now we have completed the RM registration
                session_state[RM]=OPEN;                             // so our session is now officially open
        }

        if (equal(apdu,4,6,t_application_info_enq)) {               // Now the AM wants to know about us,
                memcpy(&apdu[4],&t_application_info,3);             // so let's tell him!
                apdu[7] =0x0c;                                      // Lengthfield 12 bytes
                apdu[8] =0x01;                                      // Applicationtype = CA app
                apdu[9] =0x00;                                      // App manufacturer (from ETR-162)
                apdu[10]=0x01;                                      // idem
                apdu[11]=0x00;                                      // Manufacturer code
                apdu[12]=0x01;                                      // idem
                apdu[13]=0x06;                                      // Menu string length
                memcpy(&apdu[14],toplevel,6);                       // insert our toplevel menu name
                apduLength=0x14;                                    // adds to a total of 20 bytes
                send_spdu(apdu,apduLength,tc_id);                   // and spit it out
                session_state[AM]=OPEN;                             // Our registration with AM is now complete!
        }

        if (equal(apdu,4,6,t_ca_info_enq)) {                // Now the ca manager wants to know our caid's..
                memcpy(&apdu[4],&t_ca_info,3);              // so, why not:
                apdu[7]=nr_caids*2;                         // total bytecount of caid's
                for (i=0;i<nr_caids;i++)
                        memcpy(&apdu[8+(i*2)],caid[i],2);   // copy in our caid's
                apduLength=8+(nr_caids*2);                  // calculate total number of bytes
                send_spdu(apdu,apduLength,tc_id);           // and spit it out
                session_state[CA]=OPEN;                     // CA registration complete, we are operational!!
        }

        if (equal(apdu,4,6,t_ca_pmt)) {                             // This is where it gets interesting!
                memcpy(&apdu[4],&t_ca_pmt_reply,3);                 // Answer to ca_pmt (for now: nothing)
                apdu[7]=0x04;                                       // Lengthfield, now 4 bytes
                apdu[11]=0x73;                                      // Program 0 cannot be descrambled, technical
                apduLength=12;                                      // Total of 12 bytes
                send_spdu(apdu,apduLength,tc_id);                   // send it
        }

        if (equal(apdu,4,6,t_enter_menu)) {                         // Host wants us to enter our menu
                request_session_resource(resource[MMI]);            // So we start by opening a session to mmi
        }                                                           // Session open response sets mmi mode

        if (equal(apdu,4,6,t_display_reply)) {                      // mmi mode set, send main menu
                if (apdu[8 + lengthbytes] == 0x01) {                // If display_reply_id = 1 = mmi_mode_ack
                        session_state[MMI]=OPEN;                    // Our MMI session is now officially open
                        menu=MENU_MAIN;
Switch to Main menu
                        send_menu(menu,apdu,apduLength,tc_id);      // send the menu
                }
        }

        if (equal(apdu,4,6,t_menu_answ)) {

                if (apdu[8] == 0x00) {                      // The escape we can program here. Always return
                        if (menu == MENU_MAIN) {            // to previous menu, or exit MMI if we are in main
                                memcpy(&apdu[4],t_close_mmi,3);     // Close MMI tag, maintain session header
                                apdu[7]=0x01;                       // lengthfield = 1;
                                apdu[8]=0x00;                       // close_mmi_cmd_id = direct
                                apduLength=9;                       // Total of 8 bytes
                                send_spdu(apdu,apduLength,tc_id);
                        }
                        else {
                                menu=MENU_MAIN;                                 // Return to main menu
                                send_menu(menu,apdu,apduLength,tc_id);      // Send main menu
                        }
                }
                else {
                        switch (menu) {                     // Handling of selected choices in (sub)menu's
                                case MENU_MAIN:             // Main menu
                                        switch (apdu[8]) {  // choice_ref = selected option
                                                default:    // Always the same thing:
                                                        menu=apdu[8];
```

```
                                                        // send selected menu, if you want something else
                                                send_menu(menu,apdu,apduLength,tc_id);
                                                        // add a case for this choice..
                                        }
                                        break;
                                case MENU_CARD_INFO:            // Handling of options choosen in card info menu
                                        break;
                                case MENU_SETTINGS:             // Handling of options choosen in settings menu
                                        break;
                                case MENU_EDIT_KEYS:            // Handling of options choosen in edit keys menu
                                        break;
                                case MENU_SERIAL_UPDATE:        // Handling of options choosen in serial update menu
                                        break;
                                case MENU_CARD_UPDATE:          // Handling of options choosen in card update menu
                                        break;
                        }
                }
        }
        return OK;
}

int application_init(void) {                                    // Opens all sessions to the resources
        if (session_request_tc_id != 0x00) {                    // Do we have a transport connection?
                if (session_state[RM] == CLOSED) {              // First we have to request the Resource Manager
                        request_session_resource(resource[RM]);
                        session_state[RM]= REQUESTED;
                }
                if (session_state[RM] == OPEN && session_state[AM] == CLOSED) {
                        request_session_resource(resource[AM]);// Then, if the Resource Manager is open, we have
                        session_state[AM]= REQUESTED;           // to request the Application Manager
                }
                if (session_state[AM] == OPEN && session_state[CA] == CLOSED) {
                        request_session_resource(resource[CA]);     // Then, after the RM and the AM, we have to
                        session_state[CA]= REQUESTED;               // request the CA Manager
                }
        }
        return OK;
}

int request_session_resource(unsigned char resource[]) {    // Generic function for opening a resource

        unsigned char spdu[SPDU_LEN];                           // spdu for requesting
        long spduLength;                                        // length of spdu

        spdu[0]=OPEN_SESSION_REQUEST; spdu[1] = 0x04;           // spdu header
        memcpy(&spdu[2],resource,4);                            // copy resource in spdu
        spduLength=0x06;                                        // 6 bytes
        send_spdu(spdu,spduLength,session_request_tc_id);   // transport connection
        return OK;
}

int send_menu (unsigned char menu, unsigned char* apdu, long apduLength, unsigned char tc_id) {

        unsigned int i;                                         // counter
        long offset=0;                                          // pointer in string
        unsigned char lengthfield[3];                           // out lengthfield
        unsigned char pdu[APDU_LEN];                            // Here we will build our apdu,

        // when finished we build complete spdu in apdu

        if (strlength(menu_option[menu][0])>0) {    // Do we at least have a menu to send?
                pdu[0]=0x00;                                    // Nr. of choices, we'll add to it while building pdu
                memcpy(&pdu[1],t_text_last,3);          // Text_tag for title
                pdu[4]=strlength(toplevel);             // Length of string = main menu
                memcpy(&pdu[5],toplevel,pdu[4]);        // Copy in main menu text, always as title
                offset=5+pdu[4];                        // remember where we are in the string
                memcpy(&pdu[offset],t_text_last,3); // text_tag for subtitle
                offset+=3;
                if (menu==0) {
                        pdu[offset]=strlength(top_subheader);       // Length of subheader
                        offset++;
                        memcpy(&pdu[offset],top_subheader,pdu[offset-1]);
                                                        // For main menu we put the standard subheader
                        offset+=pdu[offset-1];
                }
                else {
```

```
                pdu[offset]=strlength(menu_option[0][menu]); // For submenu we put the name of the submenu
                offset++;
                memcpy(&pdu[offset],menu_option[0][menu],pdu[offset-1]);
                offset+=pdu[offset-1];
        }
        memcpy(&pdu[offset],t_text_last,3);                 // text_tag for subtitle
        offset+=3;
        pdu[offset]=strlength(footer); offset++;            // Standard footer always
        memcpy(&pdu[offset],footer,pdu[offset-1]);
        offset+=pdu[offset-1];
        i=0;
        while (strlength(menu_option[menu][i])>0) {
                pdu[0]++;                                   // Add an option to number of options
                memcpy(&pdu[offset],t_text_last,3);  // text_tag for menu option
                offset+=3;
                pdu[offset]=strlength(menu_option[menu][i]);        // Length of menu item
                offset++;
                memcpy (&pdu[offset],menu_option[menu][i],pdu[offset-1]);
                offset+=pdu[offset-1];
                i++;                                        // Next item
        }
        memcpy(&apdu[4],t_menu_last,3);             // copy in menu_last tag, keep session header alive
        i=calcLengthfield(offset,lengthfield);      // calculate lengthfield
        memcpy(&apdu[7],lengthfield,i);             // copy in lengthfield;
        memcpy(&apdu[7+i],pdu,offset);              // add pdu to apdu
        apduLength=7+i+offset;                      // total length of apdu
        send_spdu(apdu,apduLength,tc_id);           // send it
    }
    return OK;
}

// General purpose functions
// ========================================================================================================================
int equal(unsigned char string1[], unsigned int van, unsigned int tm, unsigned char string2[]) {
// This function compares two ranges of bytes and returns 1 if they are equal
// For example to determine a resource id field in an object
    int i;
counter
    int retval = 1;
Return value, 0=not equal, 1=equal
    for (i=van; i<=tm; i++)
        if (string1[i] != string2[i-van]) retval=0;         // one byte different? Then not equal
    return(retval);
}

int strlength(unsigned char string[]) {
// This function calculates the length of string because i can't get it to run with strlen()
    int i=0; while (string[i] !=0x00) i++;
    return (i);
}

int calcLengthfield(long length, unsigned char lengthfield[]) {
// This function calculates the Lengthfield and the number of bytes for it
// The returnvalue is the number of bytes required for the total of the lengthfield

    int lenLengthfield;

    if (length<128) {
        lenLengthfield=1;                           // Only 1 byte for lengthfield
        lengthfield[0]=length;                      // within it it's value
        return (lenLengthfield);                    // exit
    }
    if (length<256) {
        lenLengthfield=2;                           // Two byte lengthfield
        lengthfield[0]=0x81;                        // first byte indicating one seperate lengthbyte
        lengthfield[1]=length;                      // second byte contains value
        return (lenLengthfield);                    // exit
    }
    // so length >= 256
    lenLengthfield=3;                               // Three bytes lengthfield
    lengthfield[0]=0x82;                            // first byte indicating two separate lengthbytes
    lengthfield[1]=length/256;                      // First byte most significant
    lengthfield[2]=length-(length*256);             // second byte least significant
    return (lenLengthfield);
}
```

```
// Debug functions
// ================================================================================================================
#ifdef DEBUG
int scardLog (unsigned char logmsg[],long logmsgLength) {

        long i;

        for (i=0; i<logmsgLength; i++)
                scardByteWrite(SCARDs_NUM_CARD,logmsg[i]);
        scardByteWrite(SCARDs_NUM_CARD,0x0d);
        scardByteWrite(SCARDs_NUM_CARD,0x0a);
        return OK;
}

int hexconvert (unsigned char* string, unsigned char Length, unsigned char* hextstring) {
// This function puts an hex presentation in ascii format of string in hexstring for notepad logfile viewing
        unsigned int i;                                 // for-next counter
        long p=0;                                       // index in hexstring

        for (i=0; i<Length; i++){
                hexstring[p]=(string[i]/16)+48;
                if (hexstring[p]>57) hexstring[p]+=7;
                hexstring[p+1]=(string[i]-((string[i]/16)*16))+48;
                if (hexstring[p+1]>57) hexstring[p+1]+=7;
                p+=2;
        }
        return OK;
}
#endif
```

Appendix 6: Log van communicatie tussen CAM en ontvanger

Onderstaand een log van het de communicatie tussen de cam en de ontvanger volgens het EN50221 protocol. Deze logfile is gemaakt door de firmware Eva02 uit appendix 5. De cam was een Matrix Revolution en de ontvanger een Manhattan MX. De Manhattan MX reageert op verschillende plaatsen wat merkwaardig. Regels met het teken '<' is communicatie van CAM naar ontvanger en '>' van ontvanger naar CAM.

> 0100820101	t_create_t_c van de host
< 010083010180020180	c_t_c_reply
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100810101	t_rcv
< 0100A00701910400010041800201 00	Open session request Resource Manager & t_sb(no_data)
> 0100810101	Poll
< 010080020100	t_sb(no_data)
> 0100A00A01920700000100410001	Open Session Response Resource Manager, session = 0001
< 010080020100	t_sb(no_data)
> 0100A00901900200019F801000	t_profile_info_enq
< 010080020100	t_sb(no_data)
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100810101	t_rcv
< 0100A00901900200019F80110080020100	t_profile_info
> 0100A00901900200019F801200	t_profile_change
< 010080020100	t_sb(no_data)
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100A00101	Poll (?????)
< 010080020180	t_sb(data)
> 0100810101	t_rcv
< 0100A00701910400020041800201 00	Open session request session Application Manager, with t_sb(no_data)
> 0100A00A01920700000200410002	Open Session Response Application Manager, session = 0002
< 010080020100	t_sb(no_data)
> 0100A00901900200029F802000	t_application_info_enq
< 010080020100	t_sb(no_data)
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100A00101	Poll (?????)
< 010080020180	t_sb(data)
> 0100810101	t_rcv

< 0100A01501900200029F80210C01000100010645766 15F303280020100	t_application_info
> 0100810101	t_rcv (?????)
< 0100A00701910400030041800201 00	Open Session request CA Manager, with t_sb(no_data)
> 0100A00A01920700000300410003	Open Session response CA Manager, session = 0003
< 010080020100	t_sb(no_data)
> 0100A00901900200039F803000	t_ca_info_enq
< 010080020100	t_sb(no_data)
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100A00101	Poll (?????)
< 010080020180	t_sb(data)
> 0100810101	T_rcv
< 0100A00F01900200039F80310600010002000380020100	T_ca_info with t_sb(no_data)
> 0100810101	t_rcv (?????)
< 010080020100	t_sb(no_data)
> 0100A01001900200039F80320703000000000104	t_ca_pmt (stop descambling channel 0??)
< 010080020100	t_sb(no_data)
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100A00101	Poll
< 010080020180	t_sb(data)
> 0100810101	t_rcv
< 0100A00D01900200039F80330403000007380020100	t_ca_pmt_reply (we can't descramble channel 0 :-))
> 0100810101	t_rcv(?????)
< 010080020100	t_sb(no_data)
> 0100A00101	Poll
< 010080020100	t_sb(no_data)

Appendix 7: Releasenotes

Versie 1.0 (3 oktober 2004)
Document hoofdstukindeling gemaakt en hoofdstukken 1."Inleiding" en 2."Een globaal overzicht" geschreven. Appendix 1 gekopieerd van internet en appendix 2 met de releasenotes gemaakt. In deze versie zitten nog een aantal aannames van mij, waarvan ik hoop dat deze door anderen bevestigd of tegengesproken worden. Kleun ik er compleet naast, hoor ik het ook graag. Verder is de smartcard is momenteel nog een wat onderbelicht onderwerp. Misschien moet ik als gevolg hiervan de hoofdstukindeling in de toekomst herzien. Ook kan gedurende de studie blijken dat een andere volgorde wellicht handiger is. Ik ga vooralsnog verder met hoofdstuk 3. "UCAS CAM" door, in afwachting van commentaar op deze versie. Dit commentaar zal ik verwerken in versie 1.1 en hoger. In versie 2.0 zal nieuwe informatie zijn opgenomen.

Versie 2.0 (26 oktober 2004)
Geen inhoudelijke feedback ontvangen. Hoofdstuk 2 is dus ongewijzigd gebleven. Alleen het woord substream in PES veranderd. Hoofdstuk 3 "De CAM: het model" geschreven en appendix 2 en 3 tussengevoegd. Eventueel commentaar zal ik in versie 2.1 bijwerken en ondertussen ga ik verder met hoofdstuk 4.

Versie 3.0 (14 november 2004)
Hoofdstuk 4 De Matrix Cam geschreven en Appendix 4 toegevoegd. Op aanwijzing van Wildcard een kleine toevoeging over au gedaan in paragraaf 2.7. Verder de hoofdstukindeling van de komende hoofdstukken wat gewijzigd omdat het ondertussen duidelijk is geworden dat we eerst een technisch ontwerp zullen moeten maken, alvorens met programmeren te beginnen.

Versie 4.0 (26 december 2004)
In hoofdstuk 5 naar beste kunnen een technisch ontwerp voor de firmware gemaakt. Er zijn geen structurele wijzingen aangebracht in de eerdere hoofdstukken. Nu gaan we ons verdiepen in de programmeertaal C en het coderen hierin van het technisch ontwerp.

Versie 5.0 (20 maart 2005)
Ontzettend hard gewerkt aan de ontwikkeling van de firmware. Met wat hulp van Italiaanse gelijkgestemden is het uiteindelijk gelukt. Het gehele document is grondig herzien. Om juridische redenen heb ik de doelstelling van de software veranderd van een emulatie in firmware zonder emulatie. In de hoofdstukken 1 tot en met 3 heb ik aan de hand van de opgedane kennis een aantal zaken meer in detail en duidelijker omschreven. Naar aanleiding van de discussies die soms plaatsvinden over wat een CAM nu precies doet en of deze nu descrambling doet of niet, heb ik paragraaf 3.2 "*Wat doet een CAM nu precies*" geschreven. Het is dus zeker de moeite waard ook het eerste deel van het rapport dus opnieuw te lezen!

Verder heb ik hoofdstuk 5 met het technisch ontwerp totaal opnieuw geschreven. Hoewel keurig opgezet met een strikte scheiding tussen de verschillende layers, leidde de eerste versie (Eva_01) tot een programma van zo'n 1200 regels en een binary die alleen voor de verschillende buffers al meer dan 500 Kb vereiste. Het ontwerp is nu opnieuw, wat pragmatischer opgezet op basis van het door Sidsa gegeven voorbeeld voor communicatie met de pcmcia interface. Ontvangen objecten worden direct verwerkt en het totale aantal buffers is hierin van 8 tot 1 gereduceerd. Het ontwerp is ook wat globaler waardoor het kleiner is, en hierdoor ook vanuit een appendix is verhuisd naar het hoofdstuk zelf. Verder is hoofdstuk 6 Programmering geschreven en is de totale code van de firmware in appendix 5 opgenomen. In appendix 6 is ter verduidelijking een log opgenomen van de communicatie tussen CAM en ontvanger.