UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

ANTONIO AUGUSTO GIACOMELLI DE OLIVEIRA

PROPOSTA DE UM FRAMEWORK PARA O DESENVOLVIMENTO DE SOFTWARE EM CARTÕES INTELIGENTES

Porto Alegre

2019

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

ANTONIO AUGUSTO GIACOMELLI DE OLIVEIRA

PROPOSTA DE UM FRAMEWORK PARA O DESENVOLVIMENTO DE SOFTWARE EM CARTÕES INTELIGENTES

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Engenharia de Software, pelo curso de Pós-Graduação Lato Sensu em Engenharia de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Prof. Ms. Marcio Miguel Gomes

Porto Alegre

2019

Proposta de um framework para o desenvolvimento de softwares em cartões inteligentes

Antonio A. Giacomelli de Oliveira[1]

[1]Escola Politécnica – Universidade do Vale do Rio dos Sinos
Porto Alegre – RS – Brazil

antoniogiacomelli@protonmail.com

Abstract. *Although smart cards are everywere, scientific papers dealing with theirs software architecture are few. This paper proposes a software design framework for smartcards, that can mitigate software design costs over system design. The concept of hardware-dependent software is introduced, principles and design patterns focusing on reuse and extendability are presented and addressed to this engineering domain. A small card operating system is assembled using the framework components. Finally, a prototype validates the concepts.*

Key-words: smart cards, framework, software reuse, embedded systems, hardware-software co-design

Resumo. *Os cartões inteligentes (smart cards) estão presentes em diversas aplicações cotidianas. No entanto existe pouca literatura científica dedicada à arquitetura de software destes dispositivos. Este trabalho propõe um framework para o desenvolvimento de softwares para smart cards que mitiga o custo do software no projeto de um sistema desta engenharia de domínio. O conceito de software hardware-dependente é introduzido, princípios e padrões de design focados em reúso e extensibilidade são endereçados ao projeto destes sistemas. Um pequeno sistema operacional para smart cards é montado com os componentes do framework. Finalmente, os resultados em um protótipo em hardware validam os conceitos.*

Palavras-chave: smart cards, framework, reúso de software, sistemas embarcados, co-projeto hardware-software

1. Introdução

Simples cartões de memórias com mecanismo antifraude, cartões de crédito e passaportes com elevados requisitos de segurança são algumas das áreas nas quais *smart cards* são empregados. A ideia de um cartão plástico como solução para transações de consumo, substituindo o dinheiro, data dos anos 1950 [RANKL, 2007]. Os *smart cards* tal qual conhecemos hoje, tiveram sua evolução fortemente atrelada ao desenvolvimento da tecnologia microeletrônica que permitiu implementar elementos de memória e processamento em circuitos altamente integrados.

Segundo Rankl e Effing (2010) a natureza de um *smart card* depende mais do sistema operacional do cartão *(Card Operating System - COS*) que do hardware que o compõe. O projeto de software para estes dispositivos muitas vezes precisa sacrificar a flexibilidade em prol da performance e segurança, dadas as limitações de potência e memória para processamento e o atendimento às normas vigentes. A vasta aplicabilidade também leva a diferentes necessidades de projeto de software: pode-se encontrar hoje de sistemas operacionais sem nenhum grau de flexibilidade para processadores 8-bit, até sistemas que suportam várias aplicações em linguagens interpretadas, em plataformas 32-bit.

Ainda segundo Rankl (2007), em meados dos anos 80, com a evolução das tecnologias EEPROM, a indústria pôde, em algum nível, separar o aplicativo do sistema operacional: rotinas comuns eram gravadas na ROM, e a forma como estas rotinas eram chamadas e utilizadas – o aplicativo – ficava na EEPROM, que oferece alguma flexibilidade. Passados quase 40 anos do lançamento dos primeiros *smart cards* microprocessados, existem pouquíssimos COS comerciais que suportam download de aplicativos de terceiros devido às brechas de segurança que este mecanismo pode acarretar. [RANKL e EFFING, 2010].

Existem duas cadeias de negócios comuns no ramo. No chamado issuer centric, como ilustra a o *issuer* (emissor) fica responsável pelo desenvolvimento do software do cartão que é entregue ao seu cliente, sem possibilidades de alterações, como a inserção de applets. No *user centric* model (Figura 2) uma empresa interessada em fornecer soluções com smartcards, adquire os dispositivos e contrata outra (provedor de serviços) para desenvolvimento do software.

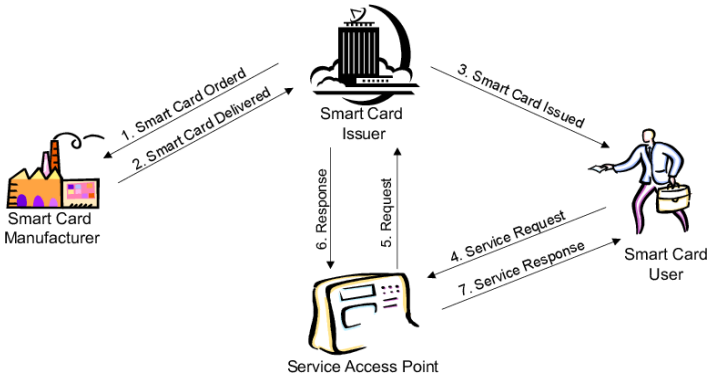

Figura 1. Modelo centrado no emissor [MAYES e MARKANTONAKIS, 2017]

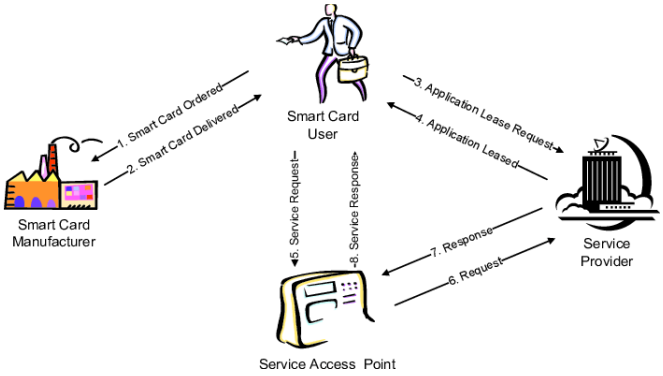

Figura 2. Modelo centrado no usuário [MAYES e MARKANTONAKIS, 2017]

Seja qual for o modelo de negócios de um provedor de serviços em *smart cards*, o co-projeto hardware-software é sempre crítico, visto que o *gap* de produtividade hardware-software vem aumentando constantemente, de forma que o custo da produção do software domina a concepção de um sistema. [WATTS, 2002].

Assim, tendo como objetivo principal aumentar a produtividade e a qualidade no ambiente de uma empresa que fornece integralmente o software para cartões inteligentes, este trabalho propõe um framework para o desenvolvimento de softwares desta engenharia de domínio que pretende mitigar o *gap* de produtividade entre hardware e software.

2. Fundamentação Teórica

2.1 Características de um *smart card*

De acordo com Mayes e Markantonakis (2017), três características classificam um dispositivo como smart card:

1) ter uma identificação exclusiva (*UID*: unique identifier);

2) ter algum elemento de segurança;

3) não ser facilmente forjado ou copiado.

A Figura 3 elenca os principais tipos de cartões comumente encontrados em soluções para o mercado. Cartões sem chip que utilizam tarjas magnéticas não são considerados smart cards e não têm relação com este trabalho. As normas ISO 7816-1/2/3 padronizam as operações baseadas em contato até a camada de transporte. As normas ISO14443 e seus derivados dedicam-se a padronização para comunicação via radio-frequência. Na camada de aplicação, os padrões são ditados pela ISO 7816-4 que define, um formato de pacote de comandos e outras normas para interoperabilidade. [RANKL e EFFING, 2010].

Além dos cartões baseados em tarja magnética os cartões podem ser divididos entre *memory cards* e *processor cards* – estes últimos são o que normalmente chamamos de *smart cards*. Os *memory cards* são aqueles comumente utilizados para tíquetes de eventos, transporte público, fidelidade em lojas, controle de acesso, vale-presentes, etc., e normalmente tem seus mecanismos implementados totalmente em hardware (não há software microprocessado).

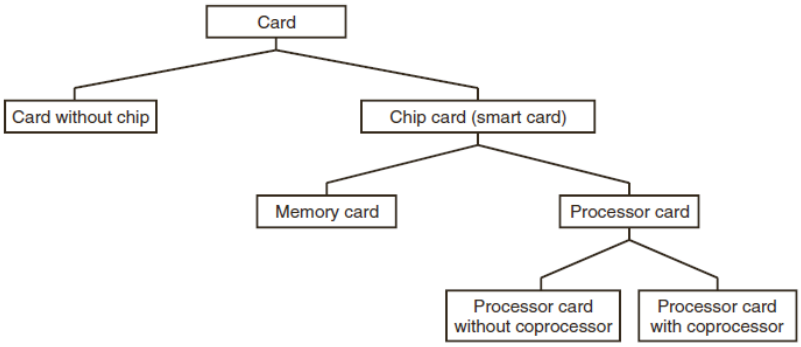

Figura 3. Tipos de cartões disponíveis no Mercado [RANKL, 2007]

Os cartões com processadores, além de conterem software também podem utilizar-se de coprocessadores numéricos para algoritmos criptográficos e são utilizados em aplicações que exigem maior segurança, de telefonia móvel à aviação civil.

A Figura 4 representa um diagrama de blocos com os principais componentes de um microcontrolador customizado para *smart cards*. As memórias não-voláteis (ROM, EEPROM ou Flash) armazenam o sistema operacional e os dados a serem utilizados. Normalmente tenta-se utilizar a memória tipo ROM o máximo possível devido ao seu baixo *footprint* (área utilizada) quando comparada a EEPROM, ou mais comumente Flash. A memória RAM, fundamental para a performance é a que ocupa maior espaço, e é a que impõe maiores limitações no compromisso área-performance. [RANKL e EFFING., 2010]

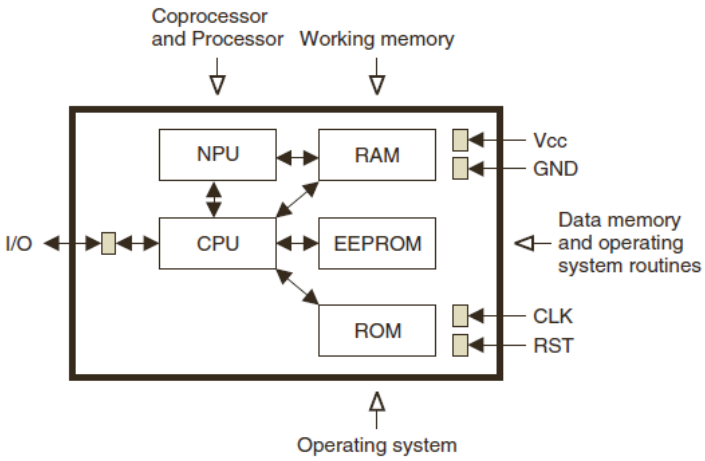

Figura 4. Diagrama de blocos de um típico smart card [RANKL, 2007]

2.2. Troca de comandos

O leitor comunica-se com um *smart card* através de um frame de bytes chamado Application Protocol Data Unit (APDU). O leitor sempre envia APDUs comando (C-APDUs) e o cartão sempre envia APDUs resposta (R-APDUs). Estes formatos estão definidos nas normas ISO7816-3 e 4. Tanto cartões de contato ou sem contato, utilizam as APDUs para enviar comandos ao aplicativo.

Uma C-APDU tem o seguinte formato:

CLA	INS	P1	P2	Nc	Data	Ne

O campo CLA define a classe da instrução a ser recebida. A mais utilizada é a classe 0x00 que define os padrões interindustriais. Outras classes indicam comandos proprietários ou especiais de segurança. O campo INS, define a instrução a ser processada. Os campos P1 e P2 são os parâmetros da instrução e variam conforme o tipo de comando. O campo Nc, indica o número de bytes que estarão contidos no campo Data. O campo Ne indica o tamanho da resposta esperada pelo leitor.

Uma R-APDU têm o seguinte formato:

Response	SW1	SW2

O campo Response é a resposta ao comando, e não é obrigatório. Os campos SW1 e SW2 formam o chamado trailer, que indicam o status do processamento. Todos definidos na norma ISO7816-4.

2.3 Design de softwares para *smart cards*

Os sistemas de software em *smart cards* evoluíram ao longo do tempo de firmwares *'bare-metal'* - aqueles sistemas embarcados monolíticos cujas rotinas operam o hardware com mínima abstração - até completos sistemas operacionais multitarefas com múltiplas aplicações. Para os propósitos deste trabalho algumas considerações devem ser feitas. As principais diferenças entre os três modelos da Figura 5 são o grau de abstração entre aplicativo e hardware e a possibilidade de acessá-lo ou não sem chamadas ao sistema operacional.

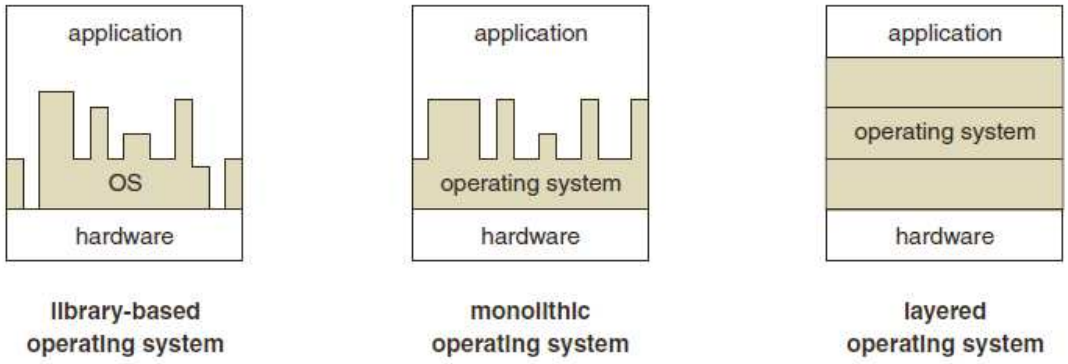

Figura 5. Arquiteturas de sistemas operacionais para para *smart card*s [RANKL e EFFING, 2010]

São chamados de *"library-based"* os sistemas operacionais embarcados em hardware que não dispõem de uma *Memory Management Unit (MMU)* para separação de privilégios de acesso a dados e recursos. Desta forma o sistema operacional e a aplicação formam um único objeto com o mesmo nível de privilégios, e as chamadas de sistema são simples chamadas de funções [ECKER, MÜLLER e DÖMER., 2009].

Na arquitetura monolítica, uma camada separa totalmente o software do hardware. Os aplicativos executam chamadas de sistema que geram exceções (interrupções em software, comumente chamadas de traps) entregando o controle ao modo supervisor, que dará ou não acesso ao serviço requisitado. [ECKER, MÜLLER e DÖMER., 2009]

A última representação refere-se aos sistemas mais modernos, multitarefas e com múltiplas aplicações, que implementam máquinas ou containers virtuais para o suporte de linguagem interpretada [GRIMAUD, LANET e VANDEWALLE., 1999]. Poucos sistemas operacionais comerciais disponibilizam linguagens interpretadas para downloads de aplicativos (applets) após o *deploy*. A principal tecnologia utilizada é o formato *JavaCard*[1]. O *MultOS*[2] é o sistema operacional comercial com linguagem interpretada mais bem estabelecido no mercado. O download de applets de terceiros é fomentado por um consórcio de empresas interessadas em criar padrões de interoperabilidade segura. As principais empresas do ramo fazem parte do consórcio *GlobalPlataform*[3].

Entretanto, existe uma enorme quantidade de *smart cards* em uso que não aceitam o download de aplicativos de terceiros. O software sai da provedora pronto para personalização ou já personalizado. Mesmo assim, estes sistemas operacionais podem suportar múltiplas aplicações isoladas uma das outras [RANKL, 2007]. É para este tipo de solução que a pesquisa do presente trabalho está direcionada.

2.4 Os microcontroladores para smart cards

Este trabalho pretende desenvolver um framework para a criação de software para smart cards e portanto não poderia deixar de abordar os elementos que compõem os computadores alvo dos softwares a serem desenvolvidos.

Os principais componentes destes computadores são o processador, os barramentos de dados e endereços, e os vários tipos de memória (RAM, ROM, EEPROM ou Flash). A capacidade de memória não-volátil nestes dispositivos tipicamente está na ordem de dezenas ou centenas de KB. O protocolo de I/O é totalmente específico para as normas deste domínio de soluções, divido em dois grandes grupos: com contato (normas ISO7816) ou sem contato (normas ISO14443). Em grande parte dos casos o I/O é do tipo UART (Universal Assynchronous Receiver Transmitter) formato de transmissão serial e assíncrono entre dois processadores, que transmite bit por bit, em uma taxa de transferência (*baudrate)* previamente determinada.

Na ponta mais alta, estão aqueles microcontroladores dotados também de processadores numéricos (NPUs) para criptografia, de MMUs para fornecer um endereçamento virtual ao programador e separar níveis de privilégio, além de múltiplas interfaces de I/O (*contact* e *contactless*) [RANKL e EFFING, 2010]

---

[1] www.oracle.com/technetwork/java/embedded/javacard/

[2] www.multos.com

[3] www.globalplataform.org

2.5 Software hardware-dependente

No projeto dos sistemas eletrônicos atuais, mais notoriamente em *systems-on-chip* (SoCs) o tempo gasto na produção de software tornou-se dominante [ECKER, MÜLLER e DÖMER., 2009]. De forma geral, o tamanho do software usado para qualquer funcionalidade está crescendo 10 vezes a cada 5 anos. [WATTS, 2002]

Aplicações embarcadas de propósito especial podem ser construídas com múltiplos cores programáveis como CPUs comuns, Processadores de sinas digitais (DSPs), e processadores de aplicação específica (ASIPs). Sistemas embarcados, tradicionalmente limitados em memória e processamento, conseguem hoje rodar aplicações mais complexas fruto do avanço no projeto e fabricação de circuitos integrados. Ora, a introdução de hardware mais complexo permite a utilização de recursos de programação também mais complexos. De fato, mais de 90% das inovações na indústria automotiva da última década são decorrência direta do desenvolvimento em software embarcado. [ECKER, MÜLLER e DÖMER, 2009].

O projeto dos dispositivos considerados computadores também envolvem projeto de software, sejam estes dispositivos de uso geral ou específico. A especialização do dispositivo leva à especialização do software e o conceito de software hardware-dependente. Segundo Ecker, Müller e Döhmer (2009), software hardware-dependente (HdS) é aquele que:

- é especialmente desenvolvido para um bloco de hardware específico: HdS é inútil sem aquele hardware

- HdS e hardware juntos, implementam uma funcionalidade: isto é, o hardware é inútil sem aquele software

- HdS provê à camada de aplicação uma interface para acessar facilmente os serviços do hardware.

2.5.1 Arquitetura de software hardware-dependente

Para todos os propósitos deste trabalho, pode-se dizer que HdS é qualquer camada de software entre a aplicação e o layer de abstração do hardware (HAL). Na Figura 6 está representado o HdS posicionado em uma arquitetura em camadas. É importante destacar que o HdS influi diretamente na funcionalidade do hardware e é mais barato de ser mantido e melhorado. [ECKER, MÜLLER e DÖMER., 2009][RANKL e EFFING, 2010]. Portanto sua reusabilidade em engenharia de domínio permite diminuir drasticamante os custos de ECOs (engineering changing orders) e NREs (non-recurring engineering costs).

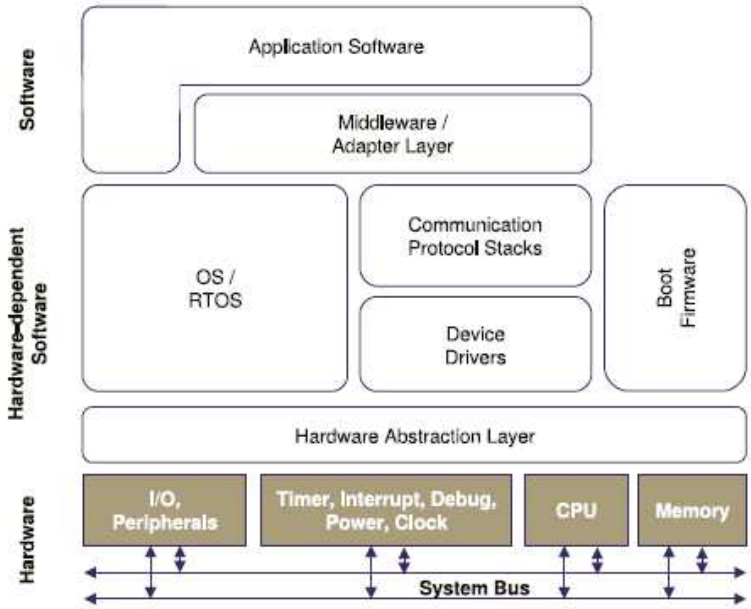

Figura 6. Arquitetura de um sistema especializado [ECKER, MÜLLER e DÖMER, 2009]

A camada de aplicação no topo é aquela que pode ser programada no nível mais alto disponibilizado. Note que em sistemas embarcados, principalmente nos chamados 'bare metal' pode ser que partes ou simplesmente toda a aplicação sejam codificadas em um nível bem mais baixo. Quando há separação entre aplicação e sistema operacional, este gerencia os serviços que o hardware pode fornecer à aplicação.

O *middleware* geralmente se refere a uma camada que adiciona serviços muito específicos, como acesso orientado a SQL por exemplo, pode ser visto como um 'adaptador'. O *communication protocol stacks*, forma a pilha de *layers* requerida para capturar os dados desde a camada de enlace de dados até a camada de aplicação (modelo OSI). O device drivers acessa a camada física, geralmente através da API proveniente de um hardware *abstraction layer*. O boot firmware é o software que coloca a camada física em estado inicial conhecido, e normalmente executa rotinas de autoteste e inicializa o sistema operacional.

2.6 Tipos de *kernel* em sistemas operacionais

Para todos os propósitos, podemos dizer que o *kernel* de um sistema operacional pode ser classificado como monolítico ou não-monolítico. Um *kernel* monolítico é aquele em que a aplicação não é autorizada a requisitar um serviço do hardware sem passar pelo próprio *kernel*. Isto cria dois domínios de memória: o *user space* da aplicação, e o *kernel space* do software supervisor.

Um *kernel* não-monolítico é aquele onde alguns (ou muitos) serviços de acesso ao hardware podem ser providos diretamente à aplicação. Estes são chamados de *microkernel, nanokernel* ou *exokernel*. Cada um deles fornece mais ou menos acesso às camadas mais baixas ao *user space*.

Na prática esta separação só funciona se houver uma MMU criando espaços virtuais de endereço (o *user space* e o *kernel space*) ou um firewall de memória. A

Figura 7 ilustra a diferença. Enquanto no kernel monolítico, o hardware é acessado unicamente através de chamadas de sistema (*system calls*), no *microkernel* os serviços mais importantes são mantidos no *kernel space*, e a aplicação acessa diretamente os serviços do sistema operacional, em algumas arquiteturas, sendo o *kernel* também um serviço. Quando o *microkernel* é muito pequeno, costuma ser chamado *nanokernel*.

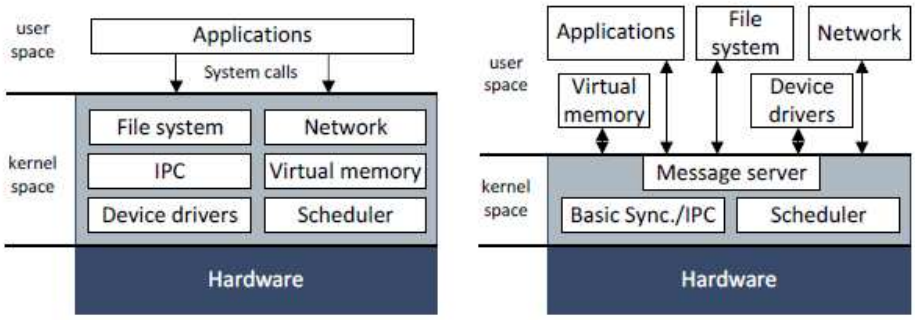

Figura 7. Kernel monolítico versus Microkernel (YU, 2010)

Pesquisadores do MIT [ENGLER e KAASHOEK, 1995] introduziram o conceito de *exokernel*, cujo lema é "retirar o máximo de abstrações de hardware possíveis" e oferecer à aplicação a oportunidade de controlar diretamente o hardware, se assim desejar.

2.6.1 Arquitetura de sistemas operacionais para *smart cards*

Os sistemas operacionais para *smart cards* não têm interface de usuário ou acesso à mídia externa. Segurança durante a execução do programa e proteção de dados são prioritários. Os sistemas mais simples, dedicados a uma aplicação específica podem ser tão pequenos quanto 10 KB. Sistemas mais complexos, como aqueles dedicados a telefonia móvel, com linguagem interpretada e suporte a *applets* ocupam algo como 500 KB de memória. [RANKL e EFFING, 2010].

Na literatura pouco se fala sobre o projeto e arquitetura destes sistemas. Se levarmos em conta as arquiteturas em camadas mostradas na Figura 5, a depender do gerenciamento de memória, todos os tipos de kernel estão ali representados. No primeiro modelo da esquerda para direita, se não houver distinção entre *user space* e *kernel space* temos na prática um sistema *"library-based"*. Se houver esta distinção, temos um *microkernel* ou *exokernel*. Os outros dois são totalmente monolíticos, sendo que o último provê mais *layers* de abstração como linguagem interpretada e máquina virtual, provavelmente.

Em [DHEM e FEYT., 2001] o autor argumenta que não há padrão para sistemas operacionais para *smart cards*, e os sistemas atuais permitem alguma portabilidade e/ou flexibilidade de software através de drivers de baixo nível sobrepostos por camadas de sistemas sob camadas de serviços, o que se assemelha ao padrão OSI (*open systems interconnect*).

Usualmente os drivers são dedicados a controlar periféricos com funcionalidade comum, mas que variam de chip para chip, como coprocessadores criptográficos, geradores de números aleatórios, escrita/leitura de memória, além de I/O (em geral uma

UART). Dependendo da aplicação há ainda sensores e outros dispositivos. Além disso, é possível distinguir duas arquiteturas gerais, nativa e *Java Card* (Figura 8).

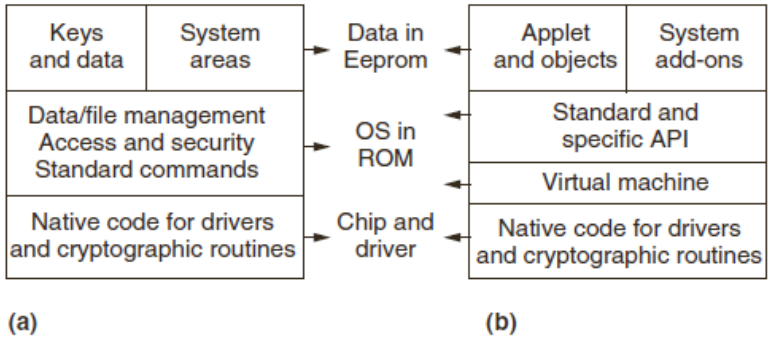

Figura 8. Arquitetura em camadas de um COS: a) Nativa b) Java Card [DHEM e FEYT, 2001]

A diferença primordial entre ambas, é que no cartão Java há uma máquina virtual que provê a possibilidade de download de código interpretado, para manejar os serviços providos pelo cartão. Este código pode ser simplesmente a personalização do cartão, ou novas aplicações de fato. Em cartões nativos, no entanto, não há outro modo de acessar os serviços, se não pelo software já nativo, i.e., a escrita de aplicativos após o cartão ser lançado é impossível.

2.7 Trabalhos relacionados

Deville et al. (2004) destaca que os softwares para *smart cards* evoluíram juntamente à engenharia de software. A necessidade de separar o sistema operacional do restante veio da necessidade de tornar o tempo de desenvolvimento de aplicativos menor. Além disso o mesmo trabalho apresenta um sistema operacional (CAMILLE) que faz uso do conceito de *exokernel* [ENGLER e KAASHOEK, 1995] com a utilização de linguagem intermediária chamada FAÇADE. O código nativo fica no *kernel space* separado do *user space*. As aplicações são escritas em linguagens de alto nível e compiladas para linguagem intermediária. Segundo o autor, cada instrução de FAÇADE, ocupa em média 3 instruções do processador com arquitetura AVR, o que tornaria o software muito próximo do hardware diminuindo os gargalos de performance que ocorrem à medida que mais níveis de abstração são adicionados.

Em [BEILKE e ROTH., 2012] é proposto uma plataforma de desenvolvimento para o estudo e pesquisa de design de *smart cards* chamada *FlexCOS*. Segundo os autores, nenhum sistema operacional de *smart cards* aberto e gratuito já tenha existido. Entretanto, pela arquitetura representada, um sistema aberto e de propósito geral, o *FreeRTOS* foi utilizado.

Além disso não há interface real compatível com ISO 7816-3 ou ISO 14443-3/4. Este autor entende que a possibilidade de implementação em hardware da camada física em lógica programável é a principal vantagem para estudo e pesquisa obtida com a utilização de um FPGA (*Field Programmable Gate Array*) – plataforma que permite a prototipação de circuitos digitais - já que o sistema operacional utilizado foi de propósito geral.

2.8 Contribuições deste trabalho

Além da inerente contribuição à comunidade acadêmica que carece de literatura no assunto, a proposta aqui apresentada complementa as contribuições dos trabalhos relacionados, descritos na seção 2.7.

Em relação à única proposta aberta para a pesquisa e desenvolvimento de softwares para *smart cards* encontrada na literatura, [BEILKE e ROTH., 2012], o trabalho aqui propõe um framework para a construção de softwares com sistema operacional dedicado à *smart cards*, enquanto no trabalho citado foi utilizado um SO de propósito geral (*FreeRTOS*). O protótipo a ser apresentado aqui também usa placa de desenvolvimento com custo muito mais baixo que uma placa baseada em FPGA, mantendo o mesmo objeto de pesquisa, que é o desenvolvimento de softwares para *smart cards.*

Em Deville et al. (2001), os autores não fornecem detalhes de arquitetura, apenas citam a utilização de *exokernel* pois a aplicação pode acessar diretamente o hardware para evitar gargalos de performance, mantendo ainda um sistema operacional em alto nível de abstração quando desejado. Em outras palavras, o *exokernel* permite extensibilidade. O autor não apresenta detalhes de implementação. Além do *exokernel*, a utilização de linguagem intermediária chamada FAÇADE é citada. De acordo com o trabalho, isto torna o aplicativo ainda mais próximo ao hardware.

O uso da linguagem intermediária com compilador próprio permite a portabilidade do software para diferentes *ISAs* (*instruction set architectures*), desde que a linguagem intermediária seja adaptada para a ISA do microprocessador que está sendo utilizado. Assim, esta abordagem faz sentido para permitir que aplicativos sejam escritos em linguagem alto nível que será compilada para a linguagem intermediária FAÇADE.

No entanto, o trabalho aqui apresentado propõe um *framework* que seja utilizado para abstrair o hardware ao máximo possível e permita agilidade no desenvolvimento do co-projeto hardware-software. Num ambiente focado em reúso e extensibilidade o acesso direto ao hardware deve ser a exceção e não a regra. Afinal, linguagens intermediárias têm baixa produtividade e reusabilidade. É preciso ainda destacar que o trabalho mencionado é de 2001. De lá para cá, uma gama de microprocessadores de baixo custo, focados em baixo consumo operam até em 80 MHz, com barramentos de 32-bit, como é o caso da família *ARM Cortex M0*. Além disso, a indústria continuou inovando na construção de processadores 8-bit, tornando-os mais eficientes e com uma vasta gama de *toolchains* (assim chamados o conjunto de ferramentas para o desenvolvimento de software embarcado em determinada plataforma).

2.9 Metodologia de Pesquisa

2.9.1 Delineamento da pesquisa

O escopo desta pesquisa é a construção de um ferramental para o co-projeto hardware-software que mitigue os custos dominantes do desenvolvimento de software na engenharia de domínio de *software nativos para smart cards.*

2.9.2 Revisão de literatura, Análise, Projeto e Validação

A pesquisa está dividida em quatro fases: *Revisão da literatura (já apresentada), Análise, Projeto e Validação*. A revisão bibliográfica foca no estudo da arquitetura de sistemas operacionais para *smart cards*, nas diferentes técnicas utilizadas no projeto de software hardware-dependente e reuso, para fundamentar os conceitos a serem utilizados nas próximas fases.

Na etapa de *Análise* a Arquitetura é proposta e modelada utilizando técnicas de engenharia de software. As escolhas arquiteturais serão justificadas à luz da revisão bibliográfica. Na fase de projeto, os componentes modelados serão prototipados em uma placa baseada em um microcontrolador *AVR ATMega328p*. Na fase de *Validação*, o *framework* será utilizado para a construção de um software nativo completo para *smart cards*, e sua funcionalidade demonstrada.

3 Arquitetura Proposta

Retomando o propósito deste trabalho que é apresentar um framework para o desenvolvimento de softwares para *smart cards* no contexto de um Provedor de Serviços que desenvolve sistemas de software nativo para estes dispositivos, esta seção detalha a proposta de arquitetura deste framework.

De forma geral os sistemas operacionais para *smart cards* trocam comandos com o leitor em frames no formato *APDU* (*Application Protocol Data Unit*), definido na norma ISO 7816-4. Estas APDUs vão e vêm de forma serial através de uma UART bidirecional. O *COS* (*Card Operating System*) é responsável por fazer o *parsing* dos comandos e controlar o acesso aos serviços que em geral incluem, I/O, sistema de arquivos e criptografia. Outra característica comum entre sistemas operacionais para *smart cards* é que o sistema de arquivos siga as normas ISO 7816-4/9.

Além disso, *smart cards* e suas principais aplicações têm quase sempre uma operação em comum: autenticação. Ou seja, para que as operações possam ser executadas, pelo menos uma entidade (leitor ou cartão) precisa verificar a procedência da outra. Arquiteturas propostas para aplicações em carteiras eletrônicas, como *Cipurse*[4] e *MIFARE*[5], exigem um método criptográfico simétrico (DES, 3DES ou AES) para autenticação mútua entre as entidades, antes que qualquer operação possa ser executada. Normalmente o sistema operacional opera um coprocessador criptográfico conforme os requisitos. Aplicações mais complexas, como cartões bancários fazem uso de criptografia assimétrica.

Assim, conforme Rankl e Effing (2009) pode-se dizer que os requisitos básicos de um COS são os seguintes:

(1) manejar a transferência de dados entre cartão e leitor;

(2) controlar a execução dos comandos;

(3) gerenciamento de arquivos;

---

[4] www.osptalliance.org

[5] www.mifare.net

(4) manejar procedimentos criptográficos, normalmente executados por hardware dedicado;

(5) executar o código de programa, gerenciando o acesso aos serviços.

3.1 Princípios de design

3.1.1 Framework de aplicações

Schmidt et al. (1997) definem um framework como: "(…) um conjunto integrado de artefatos de software (como classes, objetos e componentes) que colaboram para fornecer uma arquitetura reusável para uma família de aplicações relacionadas".

No caso deste trabalho, a família de aplicações relacionadas ao framework a ser proposto são aplicações utilizadas em *smart cards* que trocam comandos com um leitor, serialmente, no formato de *APDUs*, definida na norma ISO 7816-4. Além disso, alguns outros aspectos são comuns a essa família de aplicações como o uso de elementos seguros (criptografia ou similares) e o gerenciamento de arquivos também definidos na norma ISO 7816-4.

Este framework deve possibilitar que os aplicativos possam ser escritos mesmo antes do hardware estar definido.

3.1.2 Extensibilidade

O framework precisa poder ser estendido conforme o domínio da aplicação fica mais volumoso, ou complexo. A extensibilidade aqui pretende ser atingida utilizando os seguintes conceitos:

a) Sistema operacional com Microkernel

Quando um *microkernel* é utilizado, isto significa que serviços mais significativos como escalonamento de tarefas e tratamento de processos estarão contidos no núcleo em espaço de memória diferente do aplicativo, o *kernel space*. Os serviços restantes providos pelo sistema operacional são adicionados ao redor deste *microkernel* no user space [HERDER et al, 2006]. Estes serviços restantes podem ser vistos como *microserviços*, totalmente desacoplados no modelo cliente-servidor.

b) Programação "baseada em" e "orientada a" objetos

A linguagem C é a mais comum no projeto de sistemas embarcados e também é comum que muitas *toolchains* não suportem C++ ou outra linguagem orientada a objetos, principalmente em microcontroladores de baixo custo, que é o caso de muitos *smart cards*. Apesar de ser tradicionalmente programada de forma estruturada, pode-se escrever código orientado a objetos, suportando herança e polimorfismo [DOUGLAS, 2011], que são duas características muito desejáveis em reúso e extensibilidade, recursos fundamentais na construção desta proposta.

c) Board Support Package com a utilização de padrões de design

Os serviços de um sistema operacional são hardware-dependente, tais como interrupções, gerenciamento de energia, troca de contexto no processador, etc. Todos os elementos de hardware que precisam ser acessados pelo sistema operacional podem ser encapsulados em classes e utilizados abstraindo a interface física do hardware. Um ponto chave para reúso de software são interfaces bem definidas. O uso de padrões de

design favorece a definição precisa da camada que conecta dois pedaços de software. Padrões como Hardware Proxy e Hardware Adapter [DOUGLAS, 2011] entre outros, podem ser utilizados. A própria utilização do paradigma de orientação a objetos em ANSI-C pode ser visto como um (meta)-padrão de design.

3.2 Análise e Projeto

Nesta seção o framework será descrito e modelado em linguagem UML. A camada de aplicação poderá ser programada no nível mais alto disponibilizado, servindo-se de um sistema operacional com arquitetura baseada em micronúcleo. O micronúcleo contém o essencial para operar o sistema, como o escalonador de tarefas (SCH) e a comunicação inter-processos (IPC).

A aplicação chama os serviços no modelo cliente-servidor, que por sua vez utilizam-se do kernel também no modelo cliente-servidor. A comunicação inter-processos disponibiliza dados de um processo provedor a um processo consumidor.

O *board support package* é o conjunto de classes que adicionam mais uma camada de abstração ao HAL, aumentando a *reusabilidade* e extensibilidade dos componentes. O HAL por sua vez são aqueles códigos que trabalham diretamente com os recursos mapeados (em memória ou portas) pelo processador, como por exemplo, um *driver*.

A seguir, os componentes da arquitetura apresentados na Figura 9 serão modelados a partir de uma abordagem *top-down*.

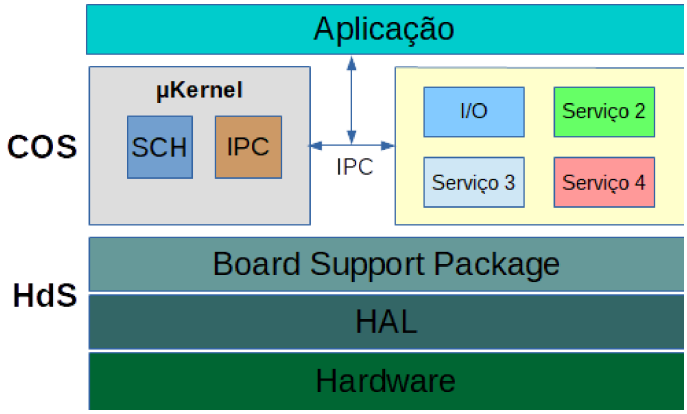

Figura 9. Arquitetura proposta, modelada em camadas

3.2.1 Aplicação

Devido à simplicidade da forma como um *smart card* troca dados com o seu leitor, serialmente e *half-duplex*, proponho que um aplicativo seja modelado no padrão de design, composto pelos seguintes objetos:

- *Parser*: vai separar os campos de classe, instrução, parâmetros da instrução, número de bytes a serem recebidos, número de bytes esperado como resposta, conforme a norma ISO 7816-4.

- *Program*: de posse das informações da APDU, vai criar os processos, que por sua vez chamarão os serviços necessários para atender ao leitor.

- *Response*: Anexa ao *payload* de resposta vindo do executor, o código de trailer indicando se a operação obteve sucesso ou não, conforme a norma ISO 7816-4.

Uma representação em diagrama de classes pode ser vista na Figura 10.

3.2.2 Microkernel

A principal função de um sistema operacional é abstrair o hardware para o desenvolvimento do aplicativo. Para tanto, é necessário que o sistema operacional controle quando cada chamada de kernel poderá acessar o hardware, i.e., ser executada – função do escalonador (*scheduler*). Além disso, é necessário haver algum tipo de comunicação inter-processos (IPC), entre um processo dito produtor e outro dito consumidor. Esta comunicação é fundamental quando *kernel space* e *user space* estão e diferentes espaços de memória.

O escalonador de tarefas proposto é um simples escalonador cooperativo no padrão RTC (*run-to-completion*). As funções são executadas até o final, na ordem em que forem alocadas, e retornam o controle para o *kernel*. Dada a natureza das aplicações para *smart cards* serem sempre do tipo "comando-resposta" com uma única interface física na maior parte dos casos, a preempção é desnecessária, já que o cartão não pode receber outro comando antes de responder ao atual. Além de ser desnecessária na maior parte das aplicações desta engenharia de domínio, o mecanismo da preempção exige troca de contexto, que varia com o processador e diminuiria a reusabilidade do *kernel*.

A comunicação inter-processos utiliza o padrão *Message Queue*. Nesta proposta, um processo é uma sub-classe do *Message Queue*, assim cada processo tem sua fila de mensagens construída no momento em que é alocado na memória. A Figura 11 mostra o diagrama de classes do *microkernel* proposto.

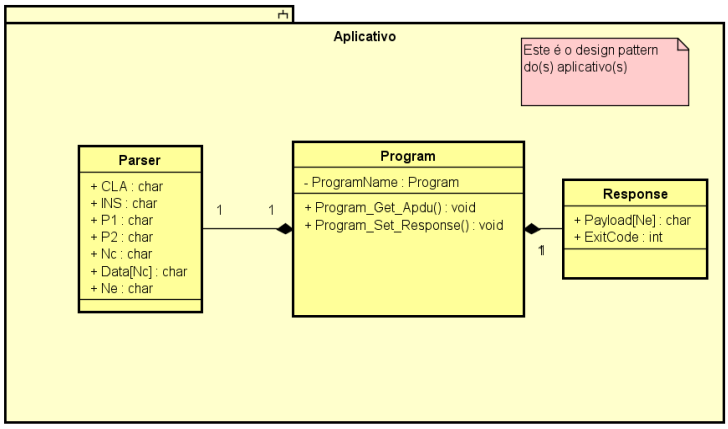

Figura 10. Diagrama de classes da camada de aplicação

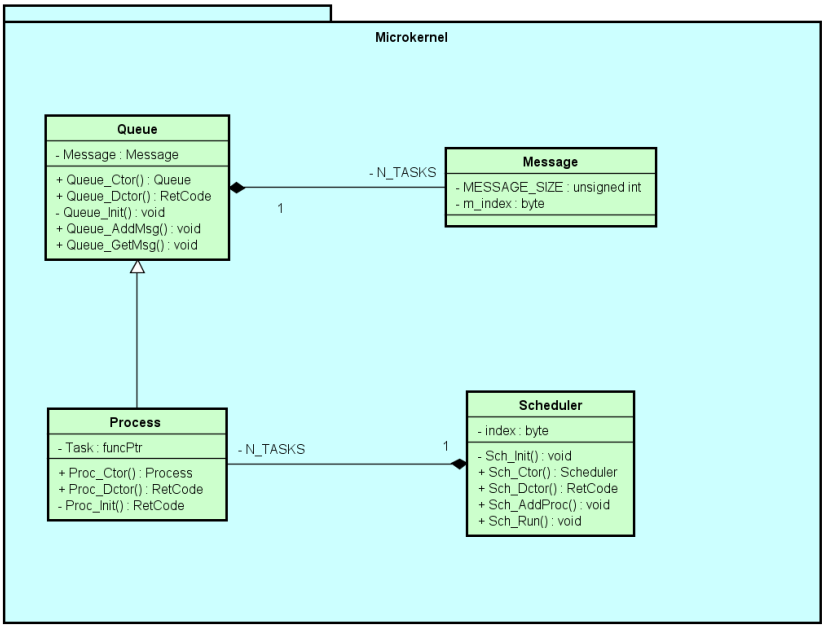

Figura 11. Diagrama de classes do Microkernel

3.2.3 Serviços

Entrada/Saída, sistema de arquivos, protocolos de comunicação, etc., são alguns exemplos de serviços providos por um sistema operacional. No caso de *smart cards*, os principais serviços são de I/O, sistema de arquivos e segurança de dados. A Figura 12 mostra o diagrama de classes de alguns serviços implementados.

O transreceptor foi implementado como uma classe virtual, que provê uma interface de ponteiros para funções que apontam para um método ou outro, em tempo de execução, ou seja: polimorfismo. Os compiladores C++ implementam polimorfismo com o mesmo conceito, o que significa que o desempenho é similar à utilização de C++ neste caso. [LIPPMAN, 1996]

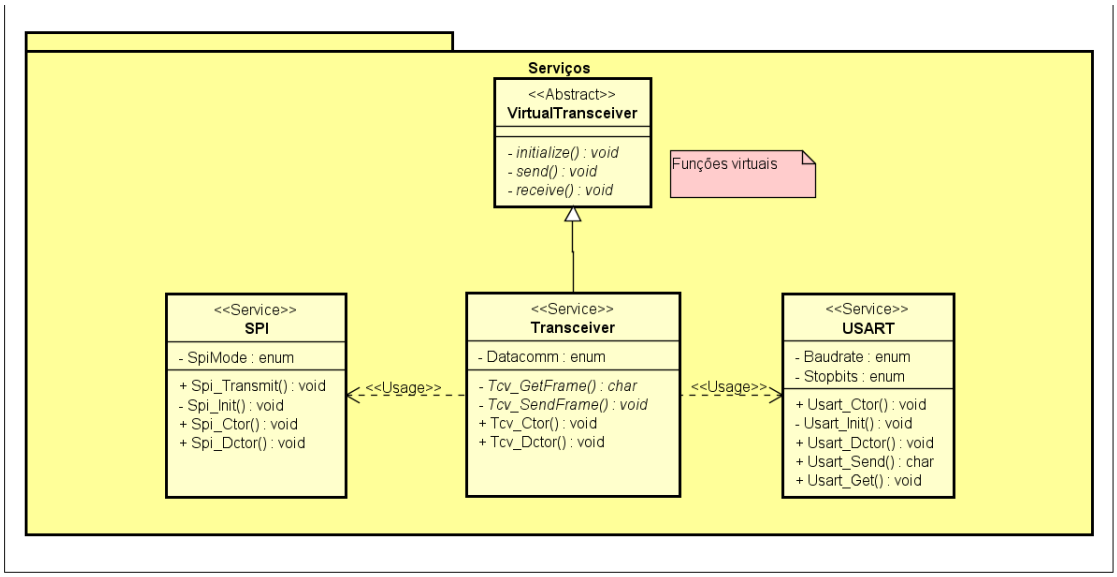

Figura 12. Diagrama de classes da camada de de se serviços

3.2.4 Board Support Package (BSP)

O BSP provê um conjunto de componentes com interface genérica o suficiente para os serviços, mas que guarda alguma relação com a arquitetura do hardware na sua interface mais baixa. Nesta proposta todos os componentes da BSP seguem o padrão de design *Hardware Proxy* [DOUGLAS, 2011]. O *proxy* conecta a camada de serviços ao *driver* e por conseguinte ao HAL, permitindo que o programador opere o hardware abstraindo todos os seus detalhes de baixo nível. Se a arquitetura do hardware a ser utilizado mudar, serão os métodos das interfaces mais baixas que serão alterados, e não sua interface com a aplicação. O BSP é por vezes chamados de HAL API [ECKER et al., 2011]. Ambas as nomenclaturas ilustram bem a sua função. A Figura 13 mostra o seu digrama de classes.

3.2.5 Hardware Abstraction Layer

O HAL são aqueles pedaços de software que fazem interface direta com o hardware e são fortemente dependentes da interface física e arquitetura dos dispositivos. Normalmente, alteram os registros da MCU para controlar periféricos mapeados em memória.

A Figura 14 mostra a controladora SPI acessando diretamente os registros do microprocessador. Vale destacar que o mapeamento destes registros a macros mais inteligíveis (*DDR_SPI, SPCR, etc.*) são de fato aquilo que chamamos de abstração de hardware. Os drivers utilizam estas abstrações.

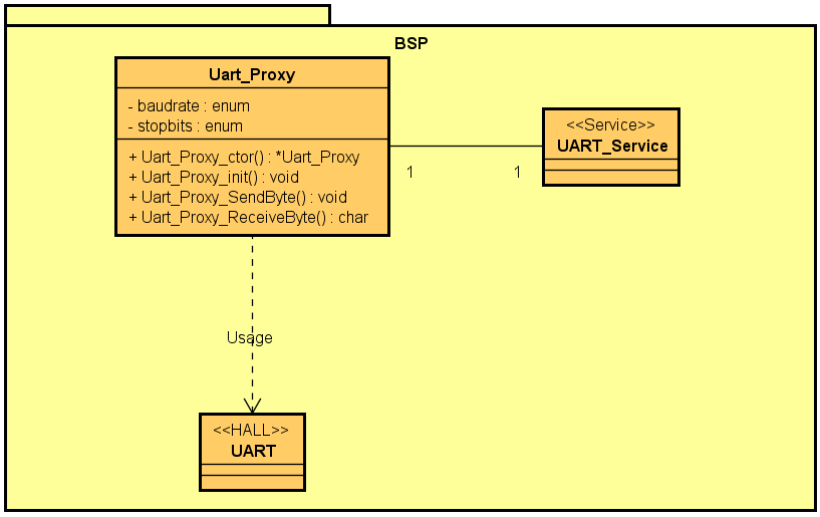

Figura 13. Diagrama de classes da proxy para UART. As demais proxies seguem o mesmo padrão.

```
/*************************************************
@file spi.c
@brief Controladora do SPI
@brief Plataforma: AVR Atmega328p


*************************************************/
```

```
#include "spi.h"
void SPI_MasterInit(void)
{
/* Configura direção dos pinos */
 DDR_SPI |= (1<<DD_MOSI)|(1<<DD_SCK)|(1<<PORTB2);
/* Habilita SPI no modo master fck/16, modo 1 */
 SPCR |= (0<<SPIE)|(1<<SPE)|(1<<MSTR)|(1<<SPR0)|(1<<CPHA);
 PORTB |= (1<<PORTB2);
}
//slave select alto
void SPI_ss_high(void)
{
PORTB |= (1<<PORTB2);
 _delay_us(1);
}
//slave select baixo
 void SPI_ss_low(void)
{
PORTB &= ~(1<<PORTB2);
 _delay_us(1);
}
// transmite e recebe
unsigned char SPI_MasterTransmit(unsigned char cData)
{
 _delay_us(1);
 /* Start transmission */
 SPDR = cData;
 /* Wait for transmission complete */
 while (!(SPSR & (1<<SPIF)));
 return SPDR;
}
```

Figura 14 Exemplo de uma controladora (driver) SPI

4. Implementação

O sistema proposto e modelado na seção 3, foi prototipado em uma placa baseada em um microcontrolador *Atmel Atmega328p*, cujo núcleo é uma CPU com arquitetura *AVR* (*RISC*) de 8-bit. A distribuição dos barramentos é do tipo *Harvard* (barramento de dados e controle distintos) o que permite que a maior parte das instruções tome apenas um ciclo de processamento.

A utilização de um processador 8-bit, longe de ser uma limitação, atende a maior parte das aplicações de um *smart card* de uso padrão, cujo processamento pesado é um serviço provido por processadores de aplicação específica (ASIPs) para criptografia.

A placa, e portanto, os periféricos disponíveis, é do modelo *Arduino Uno*. O ambiente de desenvolvimento *Atmel Studio 7.0* e a linguagem C foram utilizadas.

Esta seção apresenta as principais técnicas para a implementação do *framework* bem como sua prototipação para validação do conceito. Apenas alguns trechos de código são apresentados para ilustrar as técnicas.

4.1 Codificação

Este framework utiliza linguagem C porém utilizando um meta-padrão de design orientado a objetos. Aqui serão apresentados trechos de código do sistema que demonstram de que maneira encapsulamento, classes, herança e polimorfismo foram implementados e utilizados em ANSI-C.

4.1.1 Microkernel

Retomando o que foi escrito na seção "Análise e projeto", o *Microkernel* é composto pelas classes *Scheduler*, *Queue* e *Process*. A classe *Process* é uma subclasse de *Queue*, de forma que cada processo tem sua fila de mensagens atribuída no momento da sua construção. As classes são declaradas como estruturas de dados e encapsuladas em arquivos header, bem como os protótipos dos seus métodos.

Neste trabalho, classes e subclasses com seus métodos são declaradas em um mesmo header e codificadas em um mesmo arquivo ".c". Como aparece em destaque na Figura 15, a herança é implementada declarando a superclasse como primeiro elemento da estrutura da subclasse. Para a construção da subclasse *Process*, a superclasse *Queue* é primeiramente construída e inicializada, como aparece em destaque posteriormente na implementação.

```
/************************************************
@file process.h
@brief Header dos Processos e IPC
Superclass Queue
Subclass Process
************************************************/
#ifndef _PROCESS_H_
#define _PROCESS_H_
#include "../commondefs.h"


typedef uint8_t (*Task)();
typedef uint8_t Message;


typedef struct {
  Message message[MESSAGE_SIZE];
} Queue;


typedef struct
{
  Queue *Msg_Queue; // Superclass
  Task function;
}Process;
```

```
/*superclass' prototypes*/
Queue* Queue_Ctor(Queue* me);
void Queue_Dctor(Queue* const me);
uint8_t Queue_AddMsg(Process *process, uint8_t m_index, Message *in_message);
int8_t Queue_GetMsg(Process* process, uint8_t m_index);


/*subclass' prototypes*/
Process* Proc_Ctor(Task p_task);
void Proc_Dctor(Process* const me);


#endif //_PROCESS_H_
************************************************
@file process.c
@brief Implementação dos Processos e IPC
       Superclass Queue
       Subclass Process
************************************************/
#include "process.h"


/*******************************************
Queue Methods
*******************************************/
/******
\brief Inicializador da fila de mensagens para IPC
\param me ponteiro para endereço constante que armazena
       valor Queue
\return nao retorna valor
*************************************************************/
static void Queue_Init(Queue* const me) //private
{
  uint8_t i=0;
  for (i=0; i<MESSAGE_SIZE; i++)
  {
  me->message[i] = 0x00;
  }
}
/******
\brief Construtor da fila de mensagens
\brief Ponteiro para Queue a ser alocada na memória
\return Ponteiro para o Queue se sucesso ou ponteiro
para NULL se falhar
************************************************/
Queue* Queue_Ctor(Queue* me)
{
  me = (Queue *) malloc(sizeof(Queue));
  if (me != NULL)
  {
    Queue_Init(me);
    return me;
```

```
  }
  return me;
}


/******
\brief Construtor da fila de mensagens
\brief Ponteiro para Queue a ser desalocada na memória
\return não retorna valor
**************************************************/
void Queue_Dctor(Queue* const me)
{
 if (me != NULL)
 {
  free((void *) me);
  return;
 }
return;
}
/******
\brief Adiciona uma mensagem na fila
\param *process Ponteiro para o processo
\param m_index Indice da mensagem.
\param *in_message Ponteiro para a mensagem;
\return 0 se bem sucedido; não-zero se mal sucedido
**************************************************/
uint8_t Queue_AddMsg(Process *process, uint8_t m_index, Message *in_message)
{
 if (m_index < MESSAGE_SIZE)
 {
  process->Msg_Queue->message[m_index] = *in_message;
  return SUCCESS;
 }
 return FAIL;
}
/******
\brief Lê uma mensagem da fila
\param process Processo provedor da mensagem
\param m_index Indice da mensagem
\return 1 byte de mensagem se OK; -1 se falhar
**************************************************/
int8_t Queue_GetMsg(Process* process, uint8_t m_index)
{
 if (m_index <= MESSAGE_SIZE)
 {
  return (uint8_t) process->Msg_Queue->message[m_index];
 }
 else
 {
  return -1;
 }
```

```
}
/****************************************
Subclass Process methods
****************************************/
/******
/brief Construtor de um processo
/param Endereço da função a ser processada
/return Ponteiro para o processo
*****************************************/
Process* Proc_Ctor(Task p_task)
{
  Process* me = (Process *) malloc(sizeof(Process));
  if (me != NULL)
  {
    //constroi e inicializa superclass
    Queue* msg_queue = Queue_Ctor((Queue *) &me->Msg_Queue); //upclass
    // inicializa subclass
  me->function=p_task;
  me->Msg_Queue=msg_queue;
 }
 return me;
}
/******
/brief Desconstrutor de um processo
/param Endereço da função a ser processada
/return Não retorna valor
*****************************************/
void Proc_Dctor(Process* const me)
{
 if (me != NULL)
 {
  free(me);
 }
}
```

Figura 15. Cabeçalho e código do Microkernel

É importante notar que na construção da fila de mensagens do processo foi feito um casting para a superclasse (*upclass*). Entretanto devido ao alinhamento da memória garantido pela estrutura de dados, este casting é desnecessário na maioria dos compiladores, de forma que qualquer método da classe *Queue*, pode receber um ponteiro para *Process*. O *cast* explícito garante maior reusabilidade no entanto.

4.1.2 Serviços

Os transreceptores geralmente são do tipo *contact* ou *contactless* ou tem interface dual. Sempre que um transreceptor de diferente modelo for utilizado um novo *driver* terá que ser implantado ou desenvolvido. Além disso, há diferentes maneiras pelas quais os sinais elétricos vindos do transreceptor podem chegar até a camada de enlace de dados. Um transreceptor pode optar por utilizar uma interface *SPI* ou *I2C* para transmitir seus dados à *MCU*. Outro, pode utilizar-se de *USB*, ainda que menos comum.

Visando a reusabilidade o serviço *transceiver* foi implementado como classe abstrata para fazer o uso de funções virtuais e consequente polimorfismo. Este framework está apto a receber dados de um transreceptor via *UART* ou *SPI*, com a aplicação de polimorfismo. As funções virtuais da classe *transceiver* assumem diferentes formas a depender das configurações da subclasse (Figura 12).

Para adicionar ainda mais reusabilidade, as subclasses instanciadas do *transceiver* utilizam os serviços disponíveis à camada de aplicação (Figura 12), que por sua vez utilizam padrões do tipo proxy para abstrair a camada física do dispositivo a ser controlado (Figura 13). A Figura 16 demonstra como a classe abstrata *VirtualTransceiver* foi implementada, seus métodos e inicialização da tabela de funções virtuais *dummies*. A Figura 17 mostra como a subclasse *Transceiver* foi implementada. Destacados estão os trechos que implementam a herança e a sobrescrita da tabela de funções virtuais da Superclasse.

Vale ressaltar que esta sobrescrita acontece em tempo de execução conforme o fluxo do programa, caracterizando o polimorfismo. No caso aqui apresentado, a interface dos métodos de recepção e transmissão de frames são iguais e preveem os casos em que o transreceptor utiliza uma interface *UART* ou *SPI* para comunicar-se com a camada de enlace de dados. Este padrão de design é facilmente extensível para assumir outras formas e consequentemente atender à outras interfaces.

```
**************************************************************************/
/* @file virtualtransceiver.h */
/* @brief Classe Abstrata que oferece interface de funçoes virtuais */
/**************************************************************************/
#ifndef _VIRTUALTRANSCEIVER_H_
#define _VIRTUALTRANSCEIVER_H_
#include "../commondefs.h"
struct VirtualTransceiverVtbl; //declaração adiantada
// Classe Abstrata
typedef struct
{
  struct VirtualTransceiverVtbl const *vptr; // ponteiro para tabela de funcoes virtuais
  uint8_t framebuffer[SHARED_BUFFER_SIZE]; // framebuffer
}VirtualTransceiver;


// Tabela virtual de funções //
struct VirtualTransceiverVtbl
{
  void (*getframe)(VirtualTransceiver const * const me);
  void (*sendframe)(VirtualTransceiver const * const me);
};


// Interface do transceiver //
VirtualTransceiver* VirtualTransceiver_Ctor(VirtualTransceiver* me);
VirtualTransceiver* VirtualTransceiver_Dctor(VirtualTransceiver* const me);
```

```
// Funções com chamada virtual (late binding) //
static inline void VirtualTransceiver_GetFrame(VirtualTransceiver const * const me)
{
 (*me->vptr->getframe)(me);
}
static inline void VirtualTransceiver_SendFrame(VirtualTransceiver const * const me)
{
 (*me->vptr->sendframe)(me);
}
#endif //_VIRTUALTRANSCEIVER_H_
/*****************************************************************************/
/* @file virtualtransceiver.c */
/* @brief Implementação dos metodos da classe VirtualTransceiver */
/*****************************************************************************/
#include "../commondefs.h"
#include "virtualtransceiver.h"
#include <assert.h>


// funcoes dummy privadas
static void VirtualTransceiver_GetFrame_(VirtualTransceiver const * const me)
{
 assert(0); //erro se chamada
}
static void VirtualTransceiver_SendFrame_(VirtualTransceiver const * const me)
{
 assert(0); //erro se chamada
}
/******
\brief Construtor e inicializador
\param me ponteiro VirtualTransceiver
\return ponteiro para virtual transceiver se ok, NULL se falhar
*****************************************************************/
VirtualTransceiver* VirtualTransceiver_Ctor(VirtualTransceiver* me)
{
 uint8_t i=0;
 me = (VirtualTransceiver*) malloc(sizeof(VirtualTransceiver));
 if (me != NULL)
 {
//inicializa (static limit o escopo, const garante somente leitura)
  static struct VirtualTransceiverVtbl const vtbl =
  {
   &VirtualTransceiver_GetFrame_,
   &VirtualTransceiver_SendFrame_
  };
  me->vptr=&vtbl; // inicializei com a vtble de funções dummy
  for (i=0;i<SHARED_BUFFER_SIZE;i++) // inicializa shared buffer
  {
   me->framebuffer[i]=0x00;
  }
 }
```

```
return me; //retorna ponteiro para virtualtransceiver
}
/******
\brief Desconstrutor
\param me ponteiro VirtualTransceiver
\return não retorna valor
*****************************************************************/
void Tcv_Dctor(VirtualTransceiver const* me)
{
 if (me!= NULL)
 {
  free((void*) me);
 }
}
```

Figura 16. Cabeçalho e código do VirtualTransceiver

```
/*************************************************************************/
/* @file transceiver.h
/* @brief Subclasse da classe abstrata VirtualTransceiver */
/*************************************************************************/
#ifndef TRANSCEIVER_H_
#define TRANSCEIVER_H_
#include "../commondefs.h"
#include "virtualtransceiver.h"
/* VirtualTransceiver SubClass */
/*******************************/
typedef enum {UART, SPI} Datacomm;
typedef struct
{
 VirtualTransceiver Super; // Superclasse
 Datacomm type;
}Transceiver;
/* Interface */
Transceiver* Transceiver_Ctor(Transceiver* me, Datacomm type);
Transceiver* Transceiver_Dctor(Transceiver* const me);
#endif /* TRANSCEIVER_H_
/*************************************************************************/
/* @file transceiver.c */
/* @brief Subclasse do VirtualTransceiver */
/*************************************************************************/
#include "uart.h"
#include "transceiver.h"
#include <assert.h>
/*************************************************************************/
/* Prototipos das funcoes virtuais */
/*************************************************************************/
static void Transceiver_Send_(VirtualTransceiver const * const me);
static void Transceiver_Get_(VirtualTransceiver const * const me);
/*************************************************************************/
/* Transceiver: implementacao do metodos */
```

```
/*****************************************************************/
/******
\brief Construtor e inicializador
\param me ponteiro para endereço Transceiver
\return ponteiro para tcv_uart
*************************************************************/
Transceiver* Transceiver_Ctor(Transceiver* me, Datacomm type)
{
  uint8_t i=0;
  me=(Transceiver*) malloc(sizeof(Transceiver));
  if (me != NULL)
  {
    static struct VirtualTransceiverVtbl vtbl =
    {
     &Transceiver_Get_,
     &Transceiver_Send_
    };
 // construtor da superclasse
  VirtualTransceiver_Ctor(&me->Super);
  me->Super.vptr=&vtbl; //sobreescreve vtbl de Super
  me->type=type;
  for (i=0;i<SHARED_BUFFER_SIZE;i++)
  {
    me->Super.framebuffer[i] = i;
  }
 }
return me;
}
/*****************************************************************/
/* Implementação das funções da tabela virtual */
/*****************************************************************/
static void Transceiver_Send_(VirtualTransceiver const * const me)
{
 Transceiver * me_ = (Transceiver* )me; //downcast
 if (me_->type == UART)
 {
  Uart_send_string(me_->Super.framebuffer);
 }
 else if (me_->type == SPI)
 {
  assert(0); // erro; a ser implementado
 }
}
static void Transceiver_Get_(VirtualTransceiver const * const me)
{
  Transceiver * me_ = (Transceiver* )me;
  if (me_->type == UART)
  {
   Uart_receive_string(me_->Super.framebuffer, SHARED_BUFFER_SIZE);
  }
```

```
  else if (me_->type == SPI)
  {
    assert(0); // erro; a ser implementado
  }
}
```

Figura 17. Implementação da subclasse Transceiver

4.1.3 Board Support Package (BSP)

Para adicionar mais uma camada de abstração entre o HAL e o Sistema Operacional, possibilitando aos serviços abstração total da interface física, o padrão de design chamado Hardware Proxy (Figura 18) foi utilizado. Os chamados serviços são tratados como servidores cujo cliente é a camada de aplicação. Estes servidores por sua vez, conectam-se à camada física através de uma proxy que oferece uma interface aos drivers dos dispositivos. O programador da aplicação por sua vez inicializa, configura e opera o hardware da mesma forma para qualquer interface física a ser utilizada. Em destaque no código estão aquelas funções fortemente hardware dependentes.

```
/*****************************************************************/
/* @file uart_proxy.h
/* @brief Header da proxy para serviço UART
/*****************************************************************/
#ifndef UART_PROXY
#define UART_PROXY_H_
#include "../commondefs.h"
#include "../Services/uart.h"
typedef struct
{
 Uart_server* uart_server; // para cada proxy existe um server
}Uart_Proxy;


// Metodos
Uart_Proxy* Uart_Proxy_Ctor(Uart_Proxy* me, Uart_server* server);
uint8_t Uart_Proxy_Init(Uart_Proxy* const me);
void Uart_Proxy_Dctor(Uart_Proxy* const me);
void Uart_Proxy_SendByte(uint8_t byte);
uint8_t Uart_Proxy_ReceiveByte(void);
void Uart_Proxy_PutString(uint8_t* buffer);


#endif /* UART_PROXY_H_ */


/*************************************************
@file uart_proxy.c
@brief Componente do BSP que conecta o serviço ao
driver da Uart.
*************************************************/
```

```c
#include "../Drivers/usart.h"
#include "uart_proxy.h"


/*******************************************
Metodos Uart_Proxy
*******************************************/
/******
\brief Construtor e inicializador
\param me ponteiro para endereço da Proxy
\param server ponteiro para serviço Uart
\return ponteiro para proxy se ok, NULL se falhar
*************************************************************/
Uart_Proxy* Uart_Proxy_Ctor(Uart_Proxy* me, Uart_server* server)
{
 me = (Uart_Proxy*) malloc(sizeof(Uart_Proxy));
 if (me != NULL)
 {
  me->uart_server=server;
  Uart_Proxy_Init(me);
 }
 return me;
}
/******
\brief Desconstrutor
\param me ponteiro para endereço da proxy
\return nao retorna valor
*************************************************************/
void Uart_Proxy_Dctor(Uart_Proxy* me)
{
 if (me != NULL)
 {
  free((void *) me);
 }
}
/******
\brief Inicializa o hardware
\param me ponteiro para endereço da Proxy
\return SUCCESS ou FAIL
*********************************************************** ***/
uint8_t Uart_Proxy_Init(Uart_Proxy* const me)
{
 uint8_t r;
 r = USART_init(me); // HAL API
 return r;
}


/******
\brief Envia byte pela Uart
\param me ponteiro para endereço da Proxy
```

```
\return Não retorna parâmetro
*************************************************************/
void Uart_Proxy_SendByte(uint8_t byte)
{
 USART_send(byte); // HALL API
}


/******
\brief Recebe byte pela Uart
\param me ponteiro para endereço da Proxy
\return byte recebido
*************************************************************/
uint8_t Uart_Proxy_ReceiveByte()
{
uint8_t r;
 r = USART_get(); // HAL API
 return r;
}
void Uart_Proxy_PutString(uint8_t* StringPtr)
{
  USART_putstring((unsigned char *)StringPtr);
}
```

Figura 18. Implementação da proxy para UART

4.2. Validação do conceito

Na forma como está implantado, sem mecanismos para separação entre *kernel space* e *user space*, este *framework* trata-se, conceitualmente, de um sistema baseado em bibliotecas [ECKER, MÜLLER e DÖMER, 2009]. Neste caso, o desenvolvedor do aplicativo pode acessar as camadas mais baixas do sistema, se assim desejar. Apesar de contraproducente esta abordagem por vezes é necessária para diminuir gargalos de performance. Em um ambiente de desenvolvimento da solução completa do software de um *smart card* – contexto da proposta deste trabalho - é um padrão econômico e muito flexível. Entretanto se for requisito que uma aplicação (ou mais comumente, um *applet*) seja implantado por terceiros, este formato seria inaceitável pela vulnerabilidade.

A Tabela 1 resume as classes e métodos que de fato seriam utilizadas por um desenvolvedor da aplicação que deseja ou necessita abstrair totalmente o hardware. A assinatura das funções foi omitida por conveniência.

Classe/Estrutura	Métodos/funções
Transceiver	Transceiver_Ctor() Transceiver_Dctor() *Transceiver_Get_Frame()* *Transceiver_Send_Frame() (polimórficas)*
Program	Program_Ctor()

	Program_Dctor() Program_Get_Apdu() Program_Set_Response
Process	Process_Ctor() Process_Dctor() Queue_AddMsg() Queue_GetMsg() (herança)
Scheduler	Sch_Ctor() Sch_Dctor() Sch_AddProc() Sch_Run()
Uart_Server	Uart_Server_Ctor() Uart_Server_Dctor() Uart_send() Uart_receive() Uart_send_string() Uart_receive_string()
Uart_Proxy	Uart_Proxy_Ctor() Uart_Proxy_Dctor() Uart_Proxy_Init()

Tabela 1. Classes e métodos do framework

4.2.1 Construção do Sistema Operacional

Os *smart cards* com ou sem contato são dispositivos energizados pelo leitor (Figura 4). Nos circuitos sem contato, a antena capta um sinal de radiofrequência na faixa dos 13,56MHz, e energiza o circuito de acordo com a potência do transmissor. Na presença de vários cartões um protocolo *anticolisão* é executado, ainda na camada física.

O *kernel* aqui pretende ser utilizado na forma de um *microkernel*, um conjunto de serviços desacoplado dos demais serviços do sistema operacional. A construção do sistema operacional segue o mesmo padrão de design proposto ao aplicativo (Figura 10). E é justamente isto que diferencia um *kernel* não-monolítico de um *kernel* monolítico: o sistema operacional está contido no *User Space*.

Por dependência do hardware utilizado, o sistema operacional construído é baseado em bibliotecas. Sua função é abstrair a utilização do *microkernel* e as construções e inicialização de objetos, bem como das variáveis de controle, ao programador do *applet*. Alocação de memória, construção de objetos e comunicação inter-processos são, então, abstraídos ao programador do aplicativo. Este pequeno COS

é chamado de *BeCOS (Basic Extensible Card Operating System)* e tem a API fornecida ao programador descrita na Tabela 2.

Função	*Descrição*
osInit()	Constrói e inicializa todos os objetos e inicializa variáveis de dado e controle.
osAddApp()	Adiciona um *applet* à lista de aplicativos.
osEnableRxIRQ()	Habilita a interrupção que sinaliza byte pronto a ser lido.
osDisableRxIRQ()	Desabilita a interrupção.
osGetApdu()	Captura a APDU do *buffer* compartilhado para ser processada.
osProcessApdu()	Processa a APDU e carrega resposta ao *buffer* compartilhado.

Tabela 2. API do Sistema Operacional

Esta interface, ainda que simples, faz a utilização de todos os componentes do Framework e por consequência também é extensível. O sistema operacional pode ser simplificadamente modelado em máquina de estados representada na Figura 19. A Figura 20 mostra sua implementação em linguagem C. Apenas as instruções 0xA4 e 0x80 estão codificadas.

A Figura 21 representa a microarquitetura em componentes do sistema em software criado com este *framework.* Supondo que haja um mecanismo de separação de espaços de memória, as instâncias dos componentes estão no mesmo espaço de exceto pelo *Microkernel*. As linhas em vermelho são sinais de controle enquanto as linhas em azul são sinais de dados.

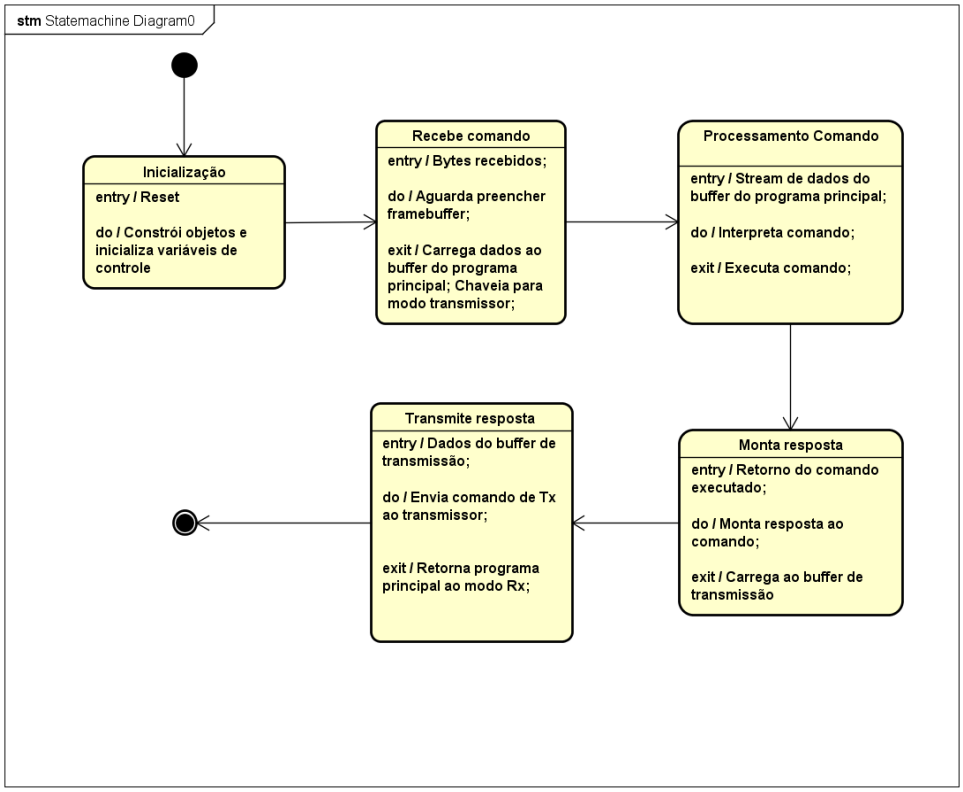

Figura 19. Máquina de estados que modela o sistema operacional

```
/*****************************************************************/
/* @file becos.c */
/* @brief Implementação do sistema operacional 'BeCOS' */
/*****************************************************************/


#include "../commondefs.h"
#include "../Components/irq.h"
#include "../Components/uart_proxy.h"
#include "../Services/transceiver.h"
#include "../Services/uart.h"
#include "../Apps/application.h"
#include "process.h"
#include "scheduler.h"
#include "becos.h"
/*****************************************************************/
/* Inicialização */
/*****************************************************************/
/* Ponteiro para função de interrupção por Rx                    */
/*****************************************************************/
Rx_IrqOn_fptr RxIrqOn;
Rx_IrqOn_fptr RxIrqOff;
/*****************************************************************/
/* Applets disponiveis */
/*****************************************************************/
```

```c
InstalledApps COSApps[N_APPS];
uint8_t SelectedApp_index;
const uint8_t App_Auth_Name[] = {0xB0, 0x50, 0xF3, 0xD3}; // nome
#define APP_AUTH_INDEX 1 // indice na lista de applets
const uint8_t App_Null_Name[] = {0xFF, 0xFF, 0xFF, 0xFF}; // nome
#define APP_NULL_INDEX 0 // indice
/**
* \brief Adiciona aplicativo a lista COSApps
*
* \param app_name Código do aplicativo. Máximo de APP_NAME_SIZE bytes
* \param app_name_size Número de bytes do código do aplicativo.
* \param app_func_call Função de chamada do aplicativo.
* \param app_index Índice do aplicativo na lista de apps instalados.
*
* \return void
*/
void osAdd_App(const uint8_t* app_name, size_t app_name_size, app_fptr app_func_call, uint8_t app_index)
{
uint8_t i;
for (i=0; i<app_name_size; i++)
{
  COSApps[app_index].App_Name[i] = app_name[i];
}
  COSApps[app_index].app_function = app_func_call;
}
/**
* \brief Inicializa a lista de aplicativos e cria um
* processo para o mesmo.
*
* \return void
*/
void osInit_App_List()
{
  osAdd_App(App_Auth_Name, 4, App_Auth, APP_AUTH_INDEX);
  osAdd_App(App_Null_Name, 4, App_Null, APP_NULL_INDEX);
  t_Select_App = App_Auth;
  p_Select_App = Proc_Ctor(t_Select_App);
}


/**
* \brief Constrói e inicializa todos os objetos e inicializa variáveis de controle.
*
* \return void
*/
void osInit()
{
  osScheduler = Sch_Ctor();
  osUartServer = Uart_server_Ctor(osUartServer,_38400_, ONE);
  osUartProxy = Uart_Proxy_Ctor(osUartProxy, osUartServer);
  osTransceiver = Transceiver_Ctor(osTransceiver, UART);
```

```
  MainProgram = Program_Ctor(MainProgram, (unsigned char*)"MainProgram");
  if (osTransceiver->type==UART)
  {
    RxIrqOn = &RXUart_IRQ_on;
    RxIrqOff = &RXUart_IRQ_off;
  }
  else
  {
   // outras IRQs
    assert(0); // nao implementado
   }
// Construção dos processos
  t_Select_App = osSelect_App;
  p_Select_App = Proc_Ctor(t_Select_App);
  t_App_Auth = App_Auth;
  p_App_Auth = Proc_Ctor(t_App_Auth)
  osInit_App_List();
  RXUart_IRQ_on();
}


/**
* \brief Habilita interrupção após recebepção de 1 byte;
* Utiliza ponteiro para função definido conforme
* a interface (UART, SPI, etc.)
* \return void
*/
void osEnableRxIRQ(void)
{
 (RxIrqOn)();
}
/**
* \brief Desabilita interrupção de Rx.
*
* \return void
*/
void osDisableRxIRQ(void)
{
  (RxIrqOff)();
}
/**
* \brief Recebe comando e armazena no buffer do programa principal
*
* \return void
*/
void osGet_Apdu()
{
  Transceiver_GetFrame(osTransceiver, SHARED_BUFFER_SIZE);
  Program_Get_Apdu(osTransceiver, MainProgram);
  osProcess_Apdu();
};
```

```
/**
* \brief Processa o comando e armazena resposta no buffer do
* programa principal.
*
* \return void
*/
void osProcess_Apdu() /* processamento de comando simplificado
para atender à validação de conceito
parâmetros como P1 e P2 não sao avaliados, por ex.
uma extensao para atender à norma ISO7816-4
de forma mais completa manteria o mesmo
padrão de design.*/
{
  uint8_t msg_index;
  switch(MainProgram->Parser->Apdu.CLA) // Classe
  {
   case 0x00:
   break; // ok, classe interindustrial suportada, segue adiante
   default: // nenhuma outra classe suportada
   Program_Set_Response(MainProgram, NULL, 0, INVALID_CLASS);
   Transceiver_SendFrame(osTransceiver, MainProgram, 0);
   break;
  }
  switch(MainProgram->Parser->Apdu.INS) // Instrução
  {
// instrucoes suportadas
   case 0xA4: // select app
   msg_index=0;
   for (msg_index=0; msg_index<(MainProgram->Parser->Apdu.Nc); msg_index++)
   {
     Queue_AddMsg(p_Select_App, msg_index, (Message*)&MainProgram->Parser->Apdu.Data[msg_index]); // envia
mensagem ao processo que vai selecionar o aplicativo
   }
   osSelect_App();
   break;
   case 0x80:
   if (SelectedApp_index==0x01)
   {
     for(msg_index=0; msg_index<(MainProgram->Parser->Apdu.Nc); msg_index++)
     {
       Queue_AddMsg(p_App_Auth, 0, (Message*)&MainProgram→Parser→>Apdu.Data[msg_index]);
     }
  Sch_AddProc(osScheduler, p_App_Auth);
 }
 break;
 default:
 Program_Set_Response(MainProgram, NULL, 0, INVALID_INS);
 Transceiver_SendFrame(osTransceiver, MainProgram, 0);
 break;
 }
}
```

```
/**
* \brief Executa instrução '0xA4' da norma ISO7816-4.
* Como não há sistema de arquivos neste protótipo
* a única interpretação possível é a de seleção de
* aplicativo.
*
* \return SUCCESS ou FAIL
*/
uint8_t osSelect_App()
{
  uint8_t i;
  uint8_t this_exit_code;
  Message thisMessage[MESSAGE_SIZE];
  for (i=0; i<(MainProgram->Parser->Apdu.Nc); i++)
  {
     thisMessage[i]=Queue_GetMsg(p_Select_App, i) // lê mensagem da fila
  }
  //varre lista de aplicativos instalados
  for (i=0; i<N_APPS; i++)
  {
    if (memcmp(thisMessage, &COSApps[i].App_Name, MainProgram->Parser->Apdu.Nc) == 0)
    {
      Program_Set_Response(MainProgram, NULL, 0, OK_DEFAULT);
      SelectedApp_index = i;
      this_exit_code = SUCCESS; // encontrou aplicativo selecionado
      break;
   }
   else
   {
     Program_Set_Response(MainProgram, NULL, 0, FILE_NOT_FOUND);
     // nao encontro aplicativo selecionado
     this_exit_code = FAIL;
    }
  }
  Transceiver_SendFrame(osTransceiver, MainProgram, 0); // para A4 responde somente o trailer
  return this_exit_code;
}
```

Figura 20. Implementação do ‘BeCOS’ em linguagem C

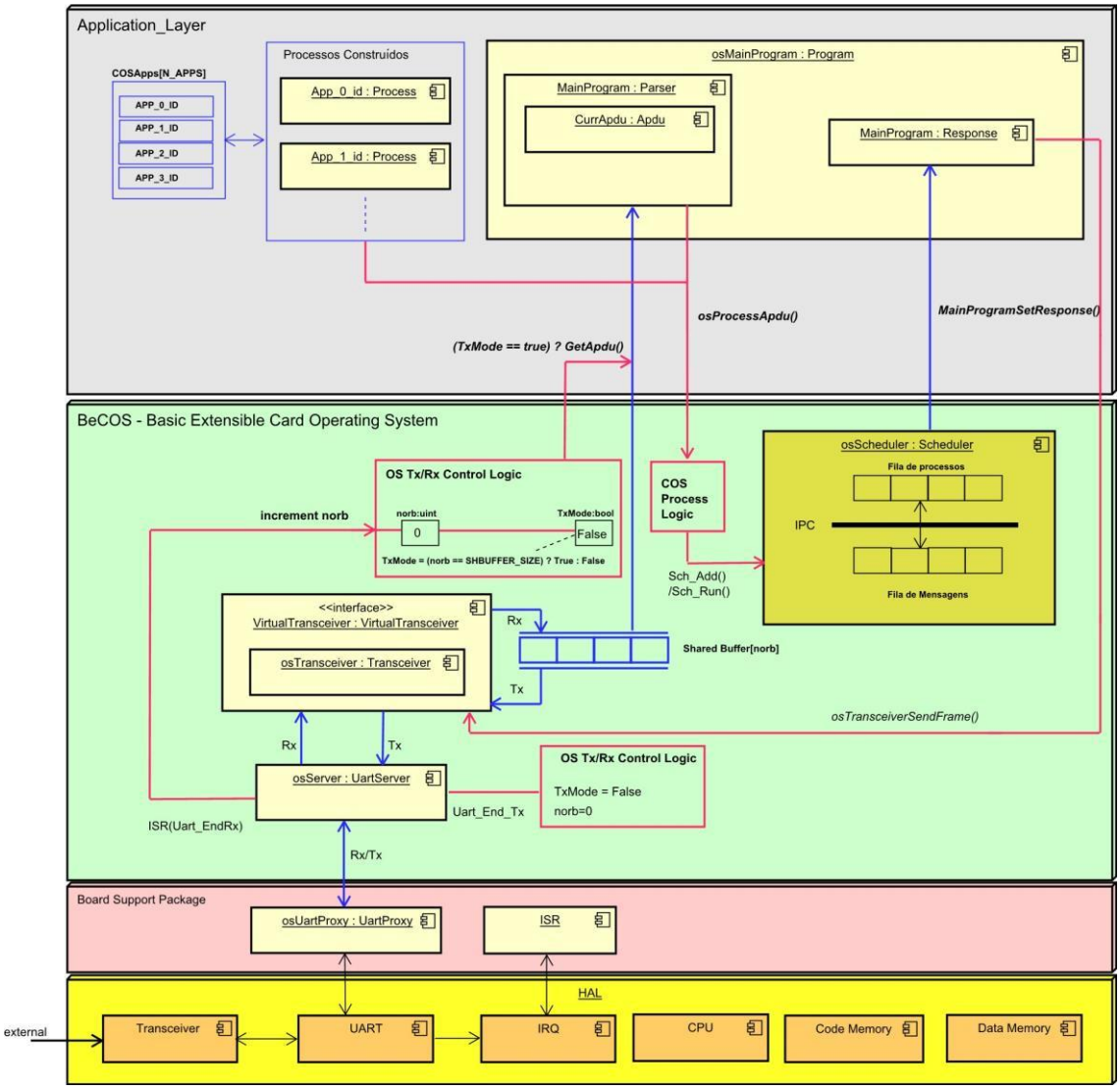

Figura 21. Microarquitetura do software construído com o framework

4.2.3 Criação de aplicativo

Na seção anterior foi mostrado a utilização do framework para a criação de um sistema operacional. Agora, a API do sistema operacional será utilizada para a criação de parte de um aplicativo. Este aplicativo executa a autenticação de um *smart card*, pelo leitor, através de uma troca de mensagens criptografadas. Os requisitos da solução são os seguintes:

1) Quando o cartão é acoplado ao leitor, seja através de interface com ou sem contato, este seleciona uma aplicação através de uma APDU, instrução '0xA4' definida na norma ISO 7816-4. O código desta aplicação é "0xB0 0x50 0xF3 0xD3".

2) Ao receber resposta de que o aplicativo foi selecionado, o leitor envia o comando 'INTERNAL AUTHENTICATE', instrução 0x80 seguido de uma seqüência aleatória de dados.

3) O cartão criptografa esta sequência aleatória de dados, utilizando mecanismo DES, e retorna a cifra ao leitor.

O leitor então descriptografa a cifra recebida e verifica se o resultado é igual ao número aleatório enviado. Em caso afirmativo, o cartão é autêntico, pois compartilha a mesma chave secreta e método criptográfico do leitor.

Este tipo de procedimento pode ser parte de soluções diversas nesta engenharia de domínio, como autorização de acesso, carteiras eletrônicas ou qualquer outro token. O arquivo *main.c* do *framework* tem o formato mostrado na Figura 22. O programador do *applet*, em princípio, nada precisa fazer no programa principal, visto que o controle da comunicação e o escalonamento das tarefas já são tratados pelo Sistema Operacional. São necessários alguns passos:

1) Adicionar nome do *applet* à lista de aplicativos do sistema, como mostra o primeiro trecho em destaque na Figura 20.
2) Criar o suporte à instrução da APDU no método *osProcessApdu*(). Se a instrução for padrão, como é o caso da instrução INTERNAL AUTHENTICATE (0x80), o suporte já pode estar codificado em um sistema operacional completo. O segundo trecho em destaque da Figura 20, mostra o suporte à instrução, provendo a mensagem com os dados a serem criptografados ao processo consumidor e o adicionando à fila de execução.
3) Criar o método do processo. A Figura 23 mostra uma implementação bastante simples, utilizando biblioteca criptográfica em software, de domínio público.

Cabe frisar que isto é apenas um exemplo de troca de mensagens entre leitor e APDU em um contexto de aplicação muito maior, que posteriormente envolveria outros processos.

```
/*****************************************************************************/
/* BeCOS Framework */
/*****************************************************************************/
#include "commondefs.h"
#include "Drivers/usart.h"
#include "Components/irq.h"
#include "Components/uart_proxy.h"
#include "Services/transceiver.h"
#include "Services/uart.h"
#include "Kernel/process.h"
#include "Kernel/scheduler.h"
#include "Kernel/becos.h"
#include "Apps/application.h"
/*
Serviço de interrupção. Interface direta com o HAL.
Em commondefs.h há as seguintes declarações:
volatile uint8_t shared_buffer[SHARED_BUFFER_SIZE];
volatile uint8_t norb;
volatile bool TxMode;
*/
```

```
ISR(USART_RX_vect) // Interface direta com o HAL
{
  PORTB=~PORTB; //toggle led indicando recebimento
  shared_buffer[norb]=USART_get();
  norb++; //incrementa número de bytes recebidos
}
int main(void)
{
  osInit();
  while(1)
  {
    while (norb < SHARED_BUFFER_SIZE)
    {
     TxMode=false;
    }
    TxMode=true;
    osGet_Apdu(); // recebe e processa o comando
   while (TxMode)
   {
     RXUart_IRQ_off();
     Sch_Run(osScheduler); // executa as tarefas agendadas pelo processamento
     osScheduler->index=0; // reinicia indice do scheduler
     TxMode=false;         // modo receptor
     norb=0;               // reinicializa norb
     RXUart_IRQ_on();
   }
 }
}
```

Figura 22. Padrão do programa principal do Framework

```
/*************************************************************************/
/* @file app_auth.c */
/* @brief Executa instrução 0x80. Criptografa dados recebidos */
/* com mecanismo DES e retorna ao leitor */
/*************************************************************************/
#include "../commondefs.h"
#include "../Drivers/usart.h"
#include "../Components/irq.h"
#include "../Components/uart_proxy.h"
#include "../Services/transceiver.h"
#include "../Services/uart.h"
#include "../Kernel/process.h"
#include "../Kernel/scheduler.h"
#include "../Kernel/becos.h"
#include "application.h"
#include "des.h" // biblioteca criptográfica de terceiros (opensource)
const uint8_t key[8] = {0xB0, 0x50, 0x15, 0xD3, 0x4D, 0x0E, 0xF3, 0xD3};
/* em um sistema comercial esta chave seria derivada a partir de uma chave
armazenada em local protegido da memória
```

```
*/
uint8_t App_Auth()
{
  const uint8_t challenge_size = MainProgram->Parser->Apdu.Nc;
  uint8_t challenge[challenge_size];
  uint8_t cypher[challenge_size];
  uint8_t msg_index = 0;
  //consome mensagem a ser criptografada
  for (msg_index=0; msg_index<(MainProgram->Parser->Apdu.Nc); msg_index++)
  {
    challenge[msg_index]=Queue_GetMsg(p_App_Auth, msg_index);
  }
  des_enc(cypher, challenge, key); // criptografa
  // coloca resposta no buffer do programa principal
  Program_Set_Response(MainProgram, cypher, 8, 0x9000);
  // requisita transmissao
  Transceiver_SendFrame(osTransceiver, MainProgram, 8);
  return 0;
}
```

Figura 23. Método para implementar a instrução 0x80

4.2.4 Prototipação e validação em hardware

A comunicação entre a camada física e a camada de enlace de dados do transreceptor sempre se dá em modo serial. Desta forma é adequado utilizar um terminal RS232 no computador para enviar comandos ao microprocessador e verificar se o software os interpreta da forma esperada. Na Figura 24 são mostrados as *APDUs* enviadas do aplicativo citado. A primeira seleciona a aplicação pelo código, e recebe como resposta 0x9000, acusando que o aplicativo foi selecionado com sucesso. Após o comando 0x80 é enviado com um campo de dados de 8 bytes, retornando estes dados criptografados também seguido do trailer 0x9000 que indica operação bem-sucedida.

Em seguida, tenta-se selecionar outro aplicativo inexistente e o trailer 0x6A82 que indica “arquivo não encontrado” é retornado. Na última linha, uma instrução não suportada foi enviada, recebendo como resposta o trailer 0x6D00, que significa “instrução inválida”. Estes *trailers* estão definidos na norma ISO 7816-4. Nesta prototipação, o tamanho do buffer do programa principal foi definido como 14 bytes. Comandos menores que 14 bytes são completados com 0x00, e serão ignorados no processamento.

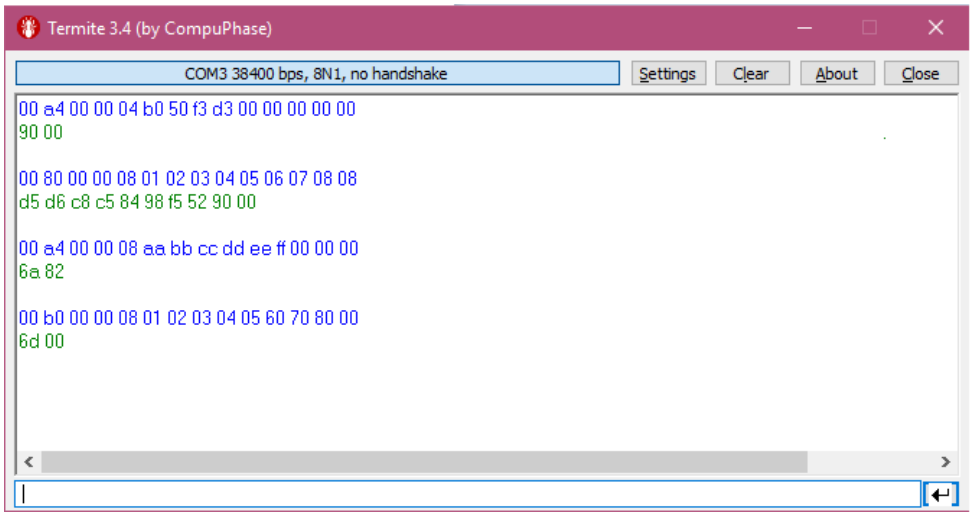

Figura 24. Comando-resposta impressos em terminal RS232

5 CONCLUSÕES

Conceitos do paradigma de programação orientado a objetos podem ser implementados, virtualmente, em qualquer linguagem se partirmos da ideia de que *classes* são estruturas de dados agregadas a um conjunto de funções que fornecem uma interface externa comum. Tanto estes *métodos* quanto estes *dados* podem ser privados ou não. No primeiro caso está o conceito de *encapsulamento*. A alocação destas estruturas com seus valores e funções, é a instância de um *objeto*. A instanciação de um objeto de uma classe dentro de outra classe implementa a *herança*. Uma *interface* que não implementa métodos pode ser utilizada para conectar-se à interface de uma outra classe, o que chamamos de *funções virtuais*. As funções virtuais podem então assumir uma forma ou outra a depender do fluxo do programa, o que caracteriza o *polimorfismo*.

Os conceitos acima implementados com os recursos da linguagem C foram utilizados neste trabalho para a construção de um *framework* para softwares no domínio de *smart cards*, na forma de um *kit* de componentes reutilizáveis e extensíveis. Entre a camada de abstração de hardware e o sistema operacional foi adicionado uma camada *proxy*. A interface mais alta da *proxy* é menos hardware-dependente que a sua interface abaixo. Assim um conjunto destes componentes forma um pacote de suporte à determinada composição de componentes físicos que necessitam de software mais hardware-dependente, o chamado *Board Support Package*. Acima fica o *Sistema Operacional* que utiliza o conceito de *Microkernel* pela sua reusabilidade, visto que na forma apresentada é desacoplado do restante do sistema.

A construção deste pequeno *COS* é uma validação de conceito do próprio *framework*. Devido ao baixo acoplamento atingido com os princípios de design utilizados (*microkernel*, orientação à objetos em ANSI-C e *board support package)* o *COS* é facilmente portável. Construiu-se parte de um aplicativo que executa um procedimento típico deste domínio de aplicação: a autenticação por uma entidade externa (leitor), através de criptografia simétrica.

O hardware utilizado não dispunha de uma *MMU* tornando-o um *COS library-based*, de forma que cada aplicativo é executado como uma *thread* do programa principal, escalonada cooperativamente. Uma eventual implementação com recursos para separação entre *kernel space* e *user space* não mudaria a arquitetura aqui apresentada, devido ao baixo acoplamento entre o *microkernel* e o restante do sistema. Desta forma, a única diferença estaria nas chamadas ao *kernel* que seriam implementadas através de um serviço de interrupção de software, e não mais através de simples chamadas de funções

Finalmente, o trabalho aqui apresentado é um kit de ferramentas reutilizáveis e extensíveis que pode ser utilizado na produção de softwares para *smartcards* permitindo que a adaptação das camadas mais fortemente hardware-dependentes possam ser desenvolvidas paralela à escrita da aplicação. Esta é uma característica muito desejável para mitigar os custos de software que - como exposto nesta pesquisa - dominam nos projetos de *SoCs*.

REFERÊNCIAS

RANKL, Wolfgang. **Smart Card Applications: Design models for using and programming smart cards**. Reino Unido: John Wiley & Sons, 2007.

RANKL, W.; EFFING, W. **Smart Card Handbook.** Reino Unido: John Wiley & Sons, 2010.

MAYES, K; MARKANTONAKIS, K. **Smart Cards, Tokens and Security Applications.** Londres: Springer, 2017.

ISO/IEC 14443 Identification cards -- Contactless integrated circuit cards -- Proximity cards. International Organization for Standardization.

ISO/IEC 7816 Identification cards – Contact integrated circuit cards. International Organization for Standardization.

ECKER, W; MÜLLER, W; DÖHMER, R. **Hardware-Dependent Software: Principles and Practice**. Londres: Springer, 2009.

GRIMAUD, G; LANET, J; VANDEWALLE, J. **FACADE: A Typed Intermediate Language Dedicated to Smart Cards.** ACM Transactions on Software Engineering Notes, 1999.

WATTS, Humphrey. **The future of software engineering: Part V.** Software Engineering Institute, 2002.

SCHMIDT, D. et al. **Object-Oriented Application Frameworks**. ACM Transactions on Software Engineering Notes, 1997

HERDER, H. et al. **Construction of a Highly Dependable Operating System.** Proceedings of the Sixth European Dependable Computing Conference, 2006

DHEM, J; FEYT, N. **Hardware and Software Symbiosis Helps Smart Card Evolution**. IEEE Micro, 2001.

KEN, Yu. RTOS **Real-Time Operating System Modelling and Simulation Using SystemC.** Tese de Doutorado, University of York, 2010.

ENGLER, D.R; KAASHOEK, M.F.. **Exterminate All Operating System Abstractions.** 5º Workshop da IEEE em Sistemas Operacionais, 1995.

DEVILLE et al., **Smart Card Operating Systems: Past, Present and Future,** 5ª Conferência, USENIX, 2004.

BEILKE, K.; ROTH, V. **FlexCOS: An Open Smartcard Platform for Research and Education**, Proceedings da 6ª Conferência Internacional em Segurança de redes e sistemas. 2012

DOUGLAS, B.P. **Design Patterns for embedded systems in C: an embedded software engineering toolkit**. Oxford: Elsevier, 2011

LIPMANN, S.B. **Inside the C++ Object Model**, Addison Wesley Longman, 1996