Inżynieria programowania kart inteligentnych

Piotr Nazimek
pnazimek@elka.pw.edu.pl

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki
Warszawa 2005

Inżynieria programowania kart inteligentnych

Książka ta przeznaczona jest dla osób pragnących poznać tematykę programowania kart procesorowych i tworzenia systemów informatycznych z ich wykorzystaniem. Zawiera ona przegląd interfejsów w różnych językach programowania oraz najpopularniejszych systemów, których głównym elementem jest karta inteligentna. Większość prezentowanych problemów została opatrzona praktycznymi przykładami implementacji.

Piotr Nazimek
pnazimek@elka.pw.edu.pl

ISBN 00-000-0000-0
wydanie pierwsze

Podziękowania

Dziękuję Panu dr. inż. Jackowi Wytrębowiczowi za pomysł oraz liczne uwagi krytyczne, zarówno podczas, jak i po zakończeniu pisania tej książki. Bezcenny był również czas jaki poświęcili mi dr Ryszard Kossowski i mgr inż. Grzegorz Wojtenko.

Dziękuję mojej Rodzinie, Przyjaciołom i Znajomym za to, że zawsze wspierają mnie w drodze, którą podążam.

Spis treści

Wstęp

Coraz częściej karty elektroniczne pojawiają się w różnych dziedzinach życia człowieka. Celem tej publikacji jest przybliżenie czytelnikowi tematyki tworzenia aplikacji z wykorzystaniem kart inteligentnych o różnych zastosowaniach: płatniczych, lojalnościowych, w infrastrukturze klucza publicznego itp.

Zakres tej publikacji obejmuje:
— krótkie wprowadzenie do tematyki związanej z kartami elektronicznymi,
— aspekty budowy fizycznej kart,
— obszary zastosowań kart,
— problematykę bezpieczeństwa podczas korzystania z kart mikroprocesorowych,
— opis bibliotek i metod programowania,
— opis środowisk dla programistów wspomagających rozwój aplikacji kartowych,
— opis środowisk symulacyjnych,
— przykłady systemów wykorzystujących karty inteligentne.

Jedną z trudności w poznawaniu tej dziedziny jest ogromna ilość skrótów. Na końcu opracowania jest zamieszczony ich wykaz.

Przyjąłem założenie, że czytelnik posiada podstawową wiedzę dotyczącą programowania strukturalnego (język C, Perl, Python), obiektowego (język Java), kryptografii oraz baz danych (język SQL).

Podczas tworzenia podręcznika wykorzystałem informacje z szeregu publikacji, zarówno książkowych jak i internetowych. Na końcu każdego rozdziału umieszczone są odnośniki do odpowiednich pozycji literatury, w których można znaleźć rozwinięcie poruszanej tematyki. Pracę ułatwiło mi także odpowiednie oprogramowanie. Rysunki sporządziłem z pomocą Xfig, a całość złożyłem w TEX z użyciem makr pakietu LATEX, z wykorzystaniem edytora Vim.

Kody źródłowe programów zawartych w tym podręczniku jak i również inne ciekawe informacje (np. łącza do publikacji i oprogramowania w Internecie) czytelnik odnajdzie na stronie `http://home.elka.pw.edu.pl/~pnazimek/smartcard/`.

Czy warto zainteresować się dziedziną jaką są karty procesorowe? Ostatnie lata są dowodem na to, że tak. Burzliwy rozwój telefonii komórkowej (karty SIM), bankowości (system płatniczy EMV), podpisu elektronicznego (infrastruktura PKI) to tylko niektóre przykłady zastosowań, w których karta inteligentna odgrywa podstawową rolę. Zwiększanie funkcjonalności kart, np. poprzez wyposażenie ich w wirtualną maszynę, wróży im zupełnie nowe zastosowania, przewyższające rolę „bezpiecznego nośnika informacji”. Rodzi też nowe problemy - już teraz ilość przeróżnych kart w naszych portfelach przyprawia o ból głowy. W każdej dziedzinie życia należy jednak zachować zdrowy rozsądek. Jestem przekonany, że dzięki lekturze tej książki, która pozwoli na poznanie systemów kartowych „od środka”, czytelnicy bez problemów zapanują nad „kartowym zawrotem głowy”.

1. Wprowadzenie

Poniższy rozdział zawiera podstawowe informacje dotyczące kart identyfikacyjnych. W sposób szczególny uwzględnieniono w nim karty elektroniczne.

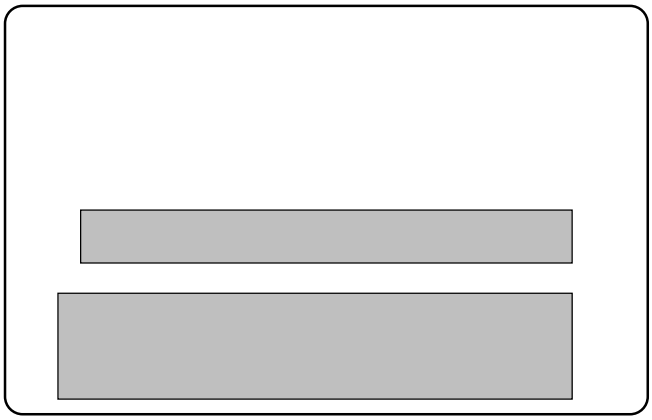
1.1. Rodzaje kart

Poniższa sekcja zawiera przegląd kart (nie tylko kart elektronicznych) opartych na formacie ID-1 określonym w normie ISO/IEC 7810. Karty identyfikacyjne wydawane są w trzech formatach (ID-1, ID-00 i ID-000), jednak najpowszechniej stosowanym jest ID-1.

1.1.1. Karty tłoczone

Tłoczenie jest najstarszą techniką umożliwiającą automatyczny odczyt danych z karty (poprzez odbicie ich na papierze). Mimo swej prostoty jest nadal powszechnie stosowane. Szczegółowy opis położenia, wielkości i wyglądu znaków na karcie zawiera norma ISO/IEC 7811.

W górnej części obszaru tłoczenia (rysunek 1) umieszcza się numer identyfikacyjny karty, który określa zarówno wydawcę jak i właściciela karty. W części dolnej umieszczane są dane właściciela jak np.: jego imię, nazwisko, adres.



Rysunek 1. Karta identyfikacyjna z zaznaczonymi obszarami tłoczenia znaków

1.1.2. Karty magnetyczne

Na karcie magnetycznej dane identyfikacyjne zapisane są na pasku magnetycznym składającym się z trzech ścieżek (rysunek 2):

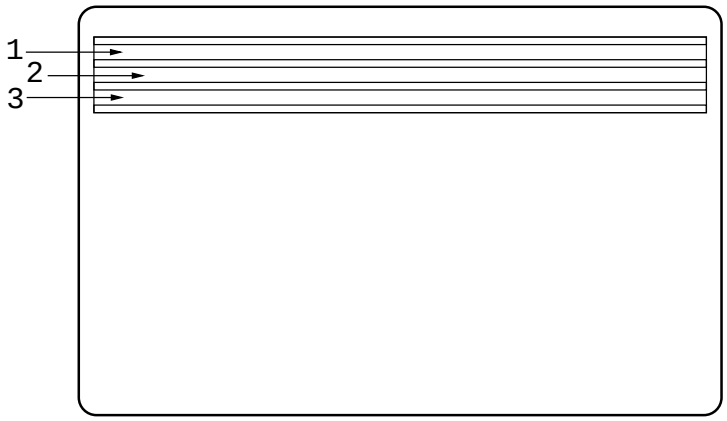
— ścieżka 1 – pojemność maksymalnie 79 znaków, 6-bitowe znaki alfanumeryczne, zapis nie jest dozwolony,
— ścieżka 2 – pojemność maksymalnie 40 znaków, 4-bitowe BCD, zapis nie jest dozwolony,
— ścieżka 3 – pojemność maksymalnie 107 znaków, 4-bitowe BCD, zapis dozwolony.

Szczegóły budowy zawarte są w normie ISO/IEC 7811.

Wadą kart magnetycznych jest możliwość przypadkowej utraty danych na skutek czynników fizycznych oraz możliwość duplikacji karty. Główną zaletą są niskie koszty produkcji zarówno kart jak i czytników.

1.1.3. Karty elektroniczne

Karta elektroniczna, a w szczególności karta inteligentna (ang. *smart card*) to kawałek plastiku najczęściej wielkości karty kredytowej wyposażonej w pamięć z chronionym dostępem.



Rysunek 2. Identyfikacyjna karta magnetyczna z zaznaczonymi ścieżkami zapisu

Sam termin *smart* oznacza, że możliwy jest jednokrotny zapis oraz kontrola dostępu do pamięci. Ponadto karta elektroniczna zawiera mikrokontroler podobny do układu Motorola 6850 lub Intel 8051.

Polskim odpowiednikiem angielskiego terminu *smart card* jest karta procesorowa (elektroniczna) zwana także inteligentną. Nie należy używać określenia *karta chipowa*. Spotykana jest także francuska nazwa *carte a puce*, czyli *karta pchły* (ponieważ małe układy na karcie wyglądają jak pchły). W terminologii angielskiej istnieje dwojaka pisownia terminu: *smart card* lub *smartcard*. Zalecany jest jednak pierwszy sposób zapisu[1].

Możemy dokonać następującej klasyfikacji kart elektronicznych:

— rozróżnienie ze względu na zastosowany układ elektroniczny:
  — karty pamięciowe (wyposażone lub nie w sprzętową ochronę danych),
  — karty mikroprocesorowe (z koprocesorem lub bez);
— rozróżnienie ze względu na rodzaj transmisji danych:
  — karty stykowe,
  — karty bezstykowe,
  — karty hybrydowe (posiadają zarówno interfejs stykowy jak i bezstykowy).

Karty pamięciowe. Karty te zawierają pamięć ROM, na której zapisane są dane identyfikacyjne użytkownika oraz pamięć EEPROM przeznaczoną dla zewnętrznych aplikacji korzystających z karty. Dodatkowo mogą być wyposażone w sprzętowy moduł chroniący dostęp do danych.

Karty mikroprocesorowe. Główną częścią tej karty jest procesor (opcjonalnie z koprocesorem) oraz pamięci ROM (system operacyjny), EEPROM (dane, kody aplikacji na karcie), RAM (pamięć operacyjna).

Karty bezstykowe. Są to karty procesorowe które umożliwiają transmisję danych bez fizycznego kontaktu z terminalem. Zastosowanie takiego rodzaju transmisji rozwiązało problem łatwego uszkodzenia karty stykowej przez przyłożenie zbyt wysokiego napięcia lub wyładowanie elektrostatyczne. Terminale, choć bardziej skomplikowane, nie są narażone na uszkodzenia fizyczne.

[1] drugi sposób zapisu jest nazwą zastrzeżoną przez kanadyjskie przedsiębiorstwo Groupmark

1.2. Karty z pamięcią optyczną

Karty takie wykorzystywane są w systemach wymagających przechowywanie dużej ilości informacji (około 4 MB). Wyposażone są w pamięć optyczną o budowie analogicznej jak na nośniku CD. Ze względu na duże koszty produkcji interfejsu obsługującego kartę (sama karta nie jest znacząco droższa niż tradycyjna karta elektroniczna) nie są obecnie w powszechnym użyciu.

1.3. Historia kart procesorowych

Wynalezienie karty elektronicznej było wynikiem ekspansji technologii elektronicznego przetwarzania danych. Oto kilka kluczowych momentów, które przyczyniły się do powstania tej dziedziny:

— **1968** (Jürgen Dethloff i Helmut Grötrupp, Niemcy) – idea karty identyfikacyjnej opartej na układzie elektronicznym,
— **1970** (Kunitaka Arimura, Japonia) – zgłasza podobny patent,
— **1974** (Roland Moreno, Francja) – zastosowanie karty w systemie płatniczym; pierwszy realny postęp w technologii kart mikroprocesorowych; ceny układów spadły co spowodowało silny rozwój technologii szczególnie we Francji i Niemczech; rozwój kart pamięciowych,
— **1979** – pierwsza karta procesorowa.

Szerokie wykorzystanie kart procesorowych (w bankowości, telekomunikacji itp.) pozytywnie wpływa na rozwój technologii. Przykładem może być implementacja maszyny wirtualnej Java na karcie elektronicznej.

1.4. Obszary zastosowań

Kierunkiem rozwoju technologii kart pamięciowych i procesorowych sterował przez cały czas obszar ich zastosowań. Obecnie typ użytej karty jest zależny od jej roli i zastosowania.

Karty pamięciowe

Ich pierwszym powszechnym zastosowaniem były karty telefoniczne. Są one bardziej odporne na podrobienie niż karty magnetyczne (które można po prostu skopiować) ponieważ zawierają układ logiczny chroniący je przed powtórnym zapisem (nie można zmienić ilości zużytych jednostek). Za przykład kolejnego zastosowania może posłużyć narodowa karta zdrowia (Niemcy). Wyposażony jest w nią każdy obywatel – w pamięci karty zawarte są jego dane wygrawerowane również laserem na powierzchni karty. Podsumowując: karty pamięciowe, mimo iż obecnie dostęp do pamięci może być sprzętowo chroniony mają ograniczoną funkcjonalność, ale ich zaletą jest niski koszt produkcji.

Karty mikroprocesorowe

Karty wyposażone w mikroprocesor po raz pierwszy zostały wykorzystane jako karty płatnicze we Francji. Rozwój tych kart był wymuszony poprzez zastosowanie ich w analogowej, a potem cyfrowej telefonii komórkowej. Teraz wykorzystywane są szeroko w różnych dziedzinach życia począwszy od prostych kart identyfikacyjnych a kończąc na kartach wyposażonych w system operacyjny z możliwością uruchamiania aplikacji na samej karcie. Na dzień dzisiejszy nie wyczerpano jeszcze wszystkich możliwości i obszarów zastosowań jakie oferuje karta mikroprocesorowa. Jest ona wciąż rozwijana zarówno w dziedzinie sprzętowej jak i aplikacyjnej.

Karty bezstykowe

Karty bezstykowe, które obsługiwane są bez kontaktu mechanicznego z interfejsem, są wykorzystywane dopiero od kilku lat. Zarówno karty pamięciowe jak i mikroprocesorowe dostępne są w wersji bezstykowej. Wykorzystywane są gdy istnieje potrzeba szybkiej identyfikacji osoby lub przedmiotu związanego z np.: kontrola dostępu, transport zbiorowy, bilety lotnicze, identyfikacja bagażu. Są jednak zastosowania, dla których jej użycie nie jest wskazane ze względów bezpieczeństwa. Przykładem może być elektroniczna portmonetka.

1.5. Organizacje standaryzujące

Ze względu ma masowe zastosowania kart elektronicznych istnieje potrzeba ich ścisłej standaryzacji. Zajmuje się nią kilka organizacji o statusie międzynarodowym. Do najważniejszych możemy zaliczyć:

— ISO (International Organization for Standardization) oraz IEC (International Electrotechnical Commission) – w ich ramach istnieje kilka grup roboczych zajmujących się między innymi: fizyczną budową kart, kartami stykowymi, kartami bezstykowymi, kartami transakcyjnymi, bezpieczeństwem, rejestracją kart i aplikacji, kartami optycznymi, metodami testowania wytrzymałości kart, metodami testowania aplikacji
— CEN (Comité Européen de Normalisation) – precyzuje standardy ISO/IEC na obszarze Europy
— ETSI (European Telecommunications Standards Institute) – rozwija normy głównie w zakresie telekomunikacji
— EMV (Europay, MasterCard, Visa) – zajmuje się standaryzacją kart wykorzystywanych w systemie płatniczym EMV
— GSM (Global System for Mobile Communications) – opracowuje standardy dotyczące kart wykorzystywanych w telefonii mobilnej
— inne organizacje np.: PC/SC Workgroup (jest to zrzeszenie największych producentów kart inteligentnych pracujące nad standardem PC/SC komunikacji z kartą w środowiskach komputerów osobistych), RSA Security (firma ta opracowała zalecenia dla kart wykorzystywanych w infrastrukturze klucza publicznego), zalecenia społeczności internetowej znane pod nazwą RFC (ang. *request for comment*)

W Polsce organizacją odpowiedzialną za ustanawianie norm jest PKN (Polski Komitet Normalizacyjny), w którego ramach utworzono KT 172 (Komisja Techniczna 172) zajmującą się kartami identyfikacyjnymi.

Uwagi bibliograficzne

Ogólne informacje dotyczące kart identyfikacyjnych (ich historia, klasyfikacje, zastosowania) przedstawione są w książkach [2, 3].

Godnymi polecenia są również serwisy internetowe [105, 100], mimo iż powiązane są głównie z zastosowaniami kart w bankowości. Warto zwrócić również uwagę na grupę dyskusyjną [99] oraz dokument [103] zawierający najczęściej zadawane pytania na tym forum.

Charakterystyki fizyczne kart identyfikacyjnych opisane są w bezpłatnej normie [6]. Kolejne części [11, 16, 22, 26, 28, 29] normy ISO/IEC 7811 zawierą szczegóły dotyczące kart magnetycznych. Publikacja [5] porusza tematykę wykorzystania kart w transakcjach finansowych.

2. Budowa kart elektronicznych

Rozdział zawiera przegląd podstawowych informacji dotyczących fizycznych i elektrycznych właściwości kart elektronicznych.

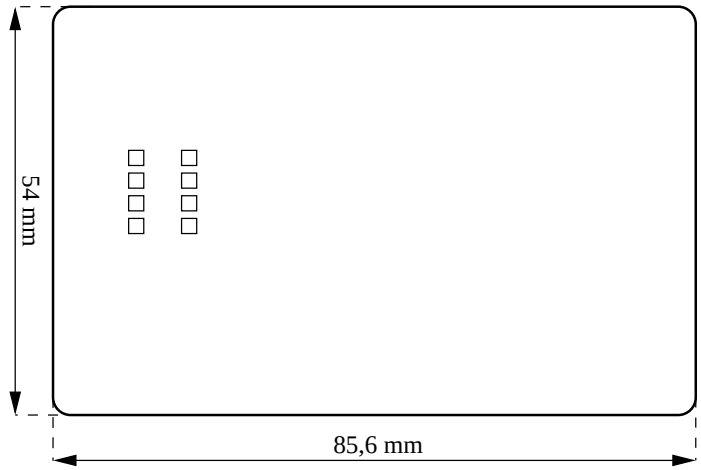
2.1. Własności fizyczne

Do właściwości fizycznych zaliczamy wymiary kart oraz wszelkiego rodzaju znaki i elementy nanoszone na karty w celach bezpieczeństwa, zwiększenia funkcjonalności lub czysto estetycznych.

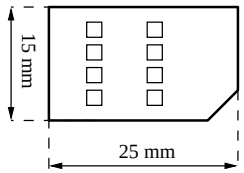
2.1.1. Wymiary kart

Na świecie spotykane są następujące formaty kart elektronicznych:

— ID-1 (rysunek 3) – wyspecyfikowany w normie ISO/IEC 7810; stosowany w kartach płatniczych, telefonicznych,
— ID-000 (rysunek 4) – stosowany w telefonii GSM,
— ID-00 (rysunek 5) – zwane również *mini-cards*.



Rysunek 3. Format ID-1

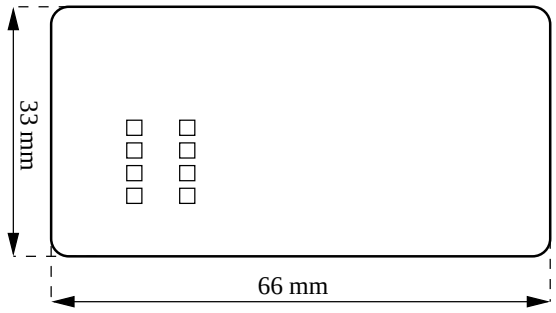


Rysunek 4. Format ID-000

Trzeba zauważyć, że często stosuje się rozwiązania polegające na fizycznym umieszczaniu karty jednego typu w drugiej (np. połączenie karty magnetycznej i elektronicznej) dzięki czemu można rozszerzyć obszar jej zastosowań lub dostosować do różnych typów terminali.

2.1.2. Wygląd karty i zabezpieczenia

Ze względu na bezpieczeństwo na kartach stosuje się szereg zabezpieczeń fizycznych:



Rysunek 5. Format ID-00

— miejsce na podpis użytkownika i jego fotografia – najprostsze z technik identyfikacji właściciela; zastosowanie odpowiedniej techniki druku zdjęcia może utrudnić podrobienie karty,
— wzory graficzne – umieszczone w różnych miejscach karty kolorowe wzory składające się z okręgów, linii i krzywych; mają zazwyczaj taką strukturę, która zapobiega prostemu kopiowaniu wzoru,
— mikrotekst – dla zwykłego użytkownika widoczne jako ciągła linia, przy silnym powiększeniu można odczytać tekst; zapobiega kopiowaniu,
— znaki ultrafioletowe – widoczne dopiero w świetle ultrafioletowym,
— hologramy – zintegrowane na stałe z kartą skutecznie uniemożliwiają jej podrobienie; nie można ich usunąć z karty w prosty sposób (łatwość uszkodzenia),
— kinetogramy – zastosowanie podobne jak do hologramów, pozwalają jednak na szybsze stwierdzenie oryginalności karty,
— laserowe wzory graficzne – niewielkie rysunki, mogące zawierać informacje o właścicielu lub wygrawerowane laserem dane właściciela karty oraz jego zdjęcie; zapobiega to podrobieniom kart, gdyż wymaga specjalistycznego sprzętu do osiągnięcia zamierzonego efektu,
— wytłaczanie – uniemożliwia np. zapisanie innych danych właściciela na karcie,
— wzory graficzne pojawiające się pod wpływem wysokiej temperatury.

Dla podniesienia poziomu bezpieczeństwa często informacje o użytych metodach fizycznych umieszczone są w pamięci karty.

2.2. Własności elektryczne

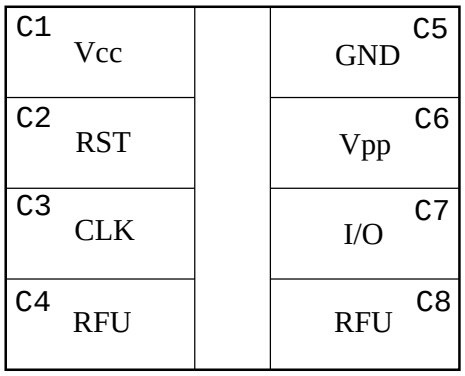
Generalnie karty zasilane są prądem o natężeniu 10 mA. Napięcie zasilania wynosi około 5 V, a średni pobór mocy to 50 mW. W przyszłości planuje się ograniczenie zużycia prądu do około 1 mA oraz zmniejszenie napięcia zasilającego[2] do 3 i 1,8 V. Jest to szczególnie ważne w telefonii komórkowej, gdzie karta SIM zasilana jest baterią. Obecnie już prawie wszystkie mikrokontrolery umieszczane na karcie posiadają tzw. tryb oszczędzania energii polegający na automatycznym odłączaniu zasilania pamięci lub innych części układu gdy są one nieużywane.

2.2.1. Styki

Na rysunku 6 przedstawiono przeznaczenie styków układu elektronicznego na karcie zgodnie z normą ISO/IEC 7816-2.

Przeznaczenie styków jest następujące:

[2] układy odpowiedzialne za wykrycie napięcia przyłożonego przez czytnik do karty mają znajdować się po stronie karty; dzięki temu nie dojdzie do sytuacji, w której stare czytniki nie będą obsługiwały nowych kart



Rysunek 6. Przeznaczenie styków na karcie (zgodnie z ISO/IEC 7816-2)

— C1 - Vcc (ang. *power supply voltage*) – napięcie zasilania, wynosi około 4,75 V - 5,25 V (dla kart wymagających napięcia 5 V),
— C2 - RST (ang. *reset*) – sygnał zerowania na który karta odpowiada „odpowiedzią na zerowanie” (zobacz 3.2),
— C3 - CLK (ang. *clock*) – przekazanie częstotliwości zegarowej z urządzenia interfejsowego; użycie jest opcjonalne (przeznaczone tylko dla kart, które nie posiadają swojego zegara wewnętrznego),
— C4 - RFU (ang. *reserved for future use*) – przeznaczony do przyszłego użycia,
— C5 - GND (ang. *ground*) – masa,
— C6 - Vpp (ang. *programming voltage*) – posiada dwa zastosowania: programowanie oraz kasowanie wewnętrznej pamięci nieulotnej,
— C7 - I/O (ang. *input/output*) – wejście/wyjście; może znajdować się w dwóch trybach: odbioru albo nadawania,
— C8 - RFU (ang. *reserved for future use*) – przeznaczony do przyszłego użycia.

Ponieważ styki C4 oraz C8 nie są aktualnie wykorzystywane zwykle nie są umieszczane przez producentów na karcie.

2.2.2. Aktywacja i dezaktywacja karty

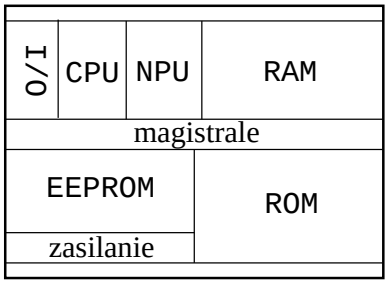
Po całkowitym mechanicznym podłączeniu styków karty do styków urządzenia interfejsowego następuje ich aktywacja. Składa się ona z następującej sekwencji:
— RST w stanie niskim,
— Vcc załączone i stabilne,
— I/O w urządzeniu interfejsowym ustawione w tryb odbioru,
— Vpp ustawione w tryb jałowy,
— CLK jeśli to konieczne podaje sygnał z odpowiednią i stabilną częstotliwością.

Jeśli karta ma być wyjęta lub nie odpowiada na sygnały interfejsu następuje jej dezaktywacja:
— RST w stanie niskim,
— CLK w stanie niskim,
— Vpp jest nieaktywne,
— I/O w stanie niskim (narzuconym przez interfejs),
— Vcc nieaktywne.

2.3. Mikrokontrolery

Mikrokontrolery stosowane na kartach procesorowych nie są typowymi układami tego rodzaju. Jest to spowodowane ich specyficznym zastosowaniem oraz dodatkowymi funkcjami jakie powinny oferować (np. zabezpieczenia). Polityka bezpieczeństwa firm, które produkują tego typu układy jest prowadzona w taki sposób by nie były one dostępne na „otwartym rynku" (częściowo wyklucza to ataki sprzętowe). W mikrokontrolerze możemy wyróżnić kilka typowych obszarów (rysunek 7):

— I/O (ang. *input/output*) – szeregowy interfejs wejścia/wyjścia,
— CPU (ang. *central processor unit*) – procesor,
— EEPROM (ang. *electrically erasable programmable read-only memory*) – programowalna pamięć nieulotna,
— NPU (ang. *numeric processing unit*) – dodatkowe układy arytmetyczne wspomagające np.: kryptografię, kompresję,
— ROM (ang. *read-only memory*) – pamięć nieulotna niemodyfikowalna,
— RAM (ang. *random-access memory*) – pamięć operacyjna,
— magistrale – zapewniają komunikację pomiędzy poszczególnymi częściami układu,
— układy zasilające.



Rysunek 7. Standardowe elementy funkcjonalne mikrokontrolera

2.3.1. Rodzaje procesorów

Przykładami procesorów stosowanych w mikrokontrolerach mogą być:

— układy z zestawem instrukcji typu CISC (ang. *complex instruction set computer*), oparte na procesorach Motorola 6805 lub Intel 8051; są to najczęściej dość proste, 8-bitowe układy,
— Hitachi H8 – 16-bitowy procesor z zestawem instrukcji opartym na RISC (ang. *reduced instruction set computer*),
— wraz z rozwojem maszyn wirtualnych (np. dla języka Java) realizowanych na kartach tworzone są układy 32-bitowe, co jednak nie oznacza, że odchodzi się od 8-bitowych procesorów (decydującą rolę gra tu cena układu); taktowane są zegarem rzędu 20 MHz.

Czołowymi producentami układów elektronicznych stosowanych na kartach inteligentnych są: Infineon, ST Microelectronics, Hitachi, Philips i Atmel.

2.3.2. Rodzaje pamięci

Wyróżniamy następujące typy pamięci stosowanych na kartach inteligentnych:

— pamięci nieulotne:
  — ROM (ang. *read-only memory*) – pamięć nieulotna niemodyfikowalna; zawiera znaczną część systemu operacyjnego, funkcje testujące i diagnostyczne,

  - EPROM (ang. *erasable programmable read-only memory*) – używany we wczesnym etapie rozwoju *smart cards*; ten rodzaj pamięci odszedł z użycia ponieważ wymaga promieniowania UV do wymazania zawartości,
  - EEPROM (ang. *electrically erasable programmable read-only memory*) – używana do przechowywania danych i aplikacji, które od czasu do czasu muszą być modyfikowane
  - Flash EEPROM (ang. *Flash read-only memory*) – w działaniu i konstrukcji są podobne do EEPROM; zasadniczą różnicą jest proces zapisu do pamięci,
  - FRAM (ang. *ferroelectric random-access memory*) – konstrukcyjnie jest podobna do EEPROM, funkcjonalnie do RAM; w przeciwieństwie do EEPROM nie wymaga jednak podwyższonego napięcia przy programowaniu i działa o wiele szybciej;
- pamięci ulotne:
  - RAM (ang. *random-access memory*) – pamięć wykorzystywana w czasie działania karty.

Pamięć typu PROM (ang. *programmable read-only memory*) nie jest używana w kartach z przyczyn bezpieczeństwa (ponieważ pamięć ta podczas programowania wymaga dostępu do magistrali, więc można ją wykorzystać do odczytania tajnych danych na karcie).

2.3.3. Dodatkowy osprzęt

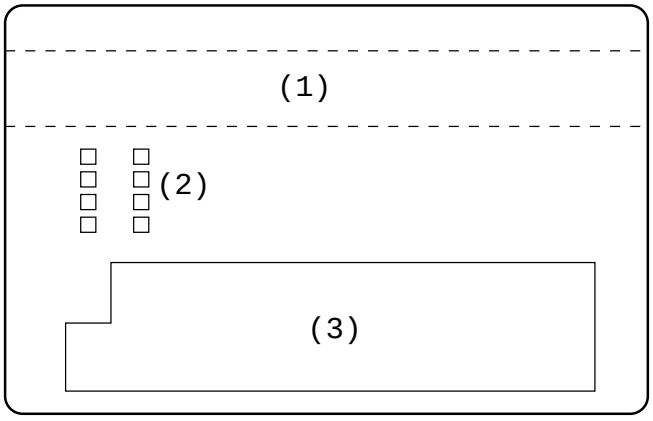
Często w mikrokontrolerach umieszczone są dodatkowe specjalizowane jednostki:

- koprocesory arytmetyczne – używane do obliczeń związanych z kluczem publicznym i podpisem elektronicznym; zawierają kilka podstawowych funkcji obliczeniowych,
- koprocesory kryptograficzne – oferują szyfrowanie algorytmem DES, 3DES, RSA lub innymi,
- generatory liczb losowych – najczęściej służą do generacji kluczy,
- układy wykrywające i korygujące błędy w pamięci EEPROM – dzięki nim można wydłużyć żywotność karty oraz podnieść poziom bezpieczeństwa i integralności danych zapisanych w pamięci; funkcje wykrywające i korygujące błędy w pamięci EEPROM są także implementowane programowo, jednak większą wydajność ma sprzętowa implementacja,
- układy wspomagające sprzętową wymianę danych – dzięki nim wymiana danych może odbywać się bez bezpośredniego udziału systemu operacyjnego; stosowane gdy wymagana jest duża szybkość transmisji (system operacyjny wymagałby podniesienia częstotliwości taktowania); realizowana przez układy UART (ang. *universal asynchronous receiver transmitter*),
- zegar wewnętrzny – pozwala osiągnąć większą moc obliczeniową mikrokontrolera i koprocesorów; niestety – im większe taktowanie tym większy pobór prądu,
- sprzętowe układy zarządzające pamięcią – są niezbędne na kartach, które umożliwiają wykonywanie załadowanego kodu; pozwalają chronić pamięć i dzielić ją dla różnych aplikacji,
- dodatkowy procesor – pozwala zwiększyć wydajność oraz zakres zastosowań karty; dwa procesory mogą pracować niezależnie, współdzieląc pamięć lub współpracować ze sobą.

2.4. Karty stykowe

W normie ISO/IEC 7816-2 szczegółowo określono położenie i wymiary styków na karcie. Ich ułożenie umożliwia tworzenie kart hybrydowych (z dodatkowym paskiem magnetycznym i wytłoczonymi danymi użytkownika). Istnieje także standard francuski, również uwzględniony w normie ISO. Różni się on położeniem styków na karcie i był wprowadzony jeszcze przed normą ISO przez AFNOR (fr. *Association Francaise de Normalisation*). W przyszłości obowiązującym standardem będzie wyłącznie ISO, ponieważ „model francuski” uniemożliwia tworzenie kart hybrydowych (pasek magnetyczny zachodzi na procesor). Położenie paska magnetycznego oraz miejsce na dane użytkownika są również szczegółowo wyspecyfikowane w normie ISO/IEC 7811. Jedna z możliwości jest schematycznie przedstawiona na rysunku 8. Pasek ma-

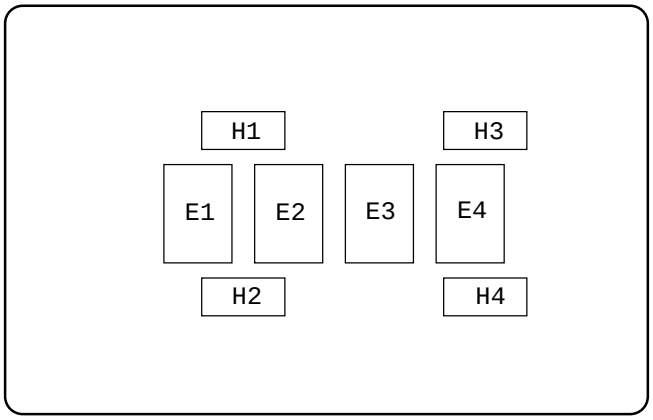
gnetyczny (1) znajduje się po przeciwnej stronie karty w stosunku do mikroprocesora (2) oraz pola z danymi użytkownika (3).



Rysunek 8. Przykładowe ułożenie paska magnetycznego, mikroprocesora i danych użytkownika na karcie

2.5. Karty bezstykowe

Odpowiednikiem styków na karcie bezstykowej są indukcyjne i pojemnościowe obszary sprzęgające (rysunek 9). Dzięki nim dochodzi do komunikacji pomiędzy CCICD (ang. *contactless integrated circuits card*, karta bezstykowa) oraz CCD (ang. *charge-coupled device*, urządzenie sprzęgające).



Rysunek 9. Położenie indukcyjnych obszarów sprzęgających na karcie bezstykowej

Komunikacja pomiędzy kartą a interfejsem odbywa się na dwa sposoby: indukcyjnie lub pojemnościowo. W danej chwili może być wykorzystywany wyłącznie jeden z wymienionych sposobów. CCD dostarcza do karty zasilanie, sygnał zegara i dane a odbiera wyłącznie dane. Zasilanie realizowane jest przez cztery indukcyjne obszary sprzęgające, które są pobudzane przez skupione, przemienne pola F1 i F2 (przesunięte względem siebie o 90°) oraz F3 i F4 (przesunięte względem siebie o 180°). Częstotliwość pola zasilającego wynosi około 5 MHz, a minimalny poziom mocy w kierunku do CCICD to około 150 mW.

Karty mogą być zasilane również przez wewnętrzną baterię (mówimy wtedy o karcie aktywnej w przeciwieństwie do modelu z dostarczanym zasilaniem - karty pasywne).

Pomiędzy kartą a czytnikiem osiągane szybkości transmisji danych to od 9600 b/s do nawet 100 kb/s.

Możemy wyróżnić następujące rodzaje kart bezstykowych:

— *close-coupling cards* (norma ISO/IEC 10536) – karty działające na odległość około 0 mm - 10 mm; stosowane głównie jako karty identyfikacyjne i dostępowe, wymagają bezpośredniego przyłożenia do czytnika,
— *remote coupling card* (normy ISO/IEC 14443 oraz ISO/IEC 15633) – działają na odległości do 1 m od terminala; używane jako np.: karty dostępowe, bilety lotnicze, identyfikatory bagażu, prawa jazdy i dowody rejestracyjne pojazdów,
— *proximity integrated circuit(s) cards* (norma ISO/IEC 14443) – zakres ich działa to około 10 mm - 10 cm; stosowane jako karty dostępowe, bilety,
— *hands-free integrated circuit(s) cards* (norma ISO/IEC 15693) – działają na odległość większą niż 1 m.

Należy zwrócić uwagę, że wiele ze standardów kart bezstykowych nie jest jeszcze ukończonych i wciąż się rozwija.

Istnieją również rozwiązania hybrydowe łączące interfejs stykowy z bezstykowym operujący na wspólnym układzie elektronicznym. Zaletą takiego rozwiązania jest zwiększenie uniwersalności karty a także poziomu bezpieczeństwa.

Uwagi bibliograficzne

Tematyka poruszna w tym rozdziale jest przedstawiona w dużo szerszym zakresie w [2, 3].

Norma [6] zawiera szczegółową specyfikację formatów i cech fizycznych kart identyfikacyjnych. Przeznaczenie styków karty procesorowej opisuje standard [15]. Normami ISO/IEC, które związane są z tematyką kart bezstykowych i zostały przetłumaczone przez PKN są [10, 18, 24]. Pozostałe standardy dotyczące tego zagadnienia, czyli [7, 13, 19, 25, 8, 14, 20], dostępne są w języku angielskim.

Tematykę testowania wytrzymałości kart zawarto w dokumentach [12, 17, 23].

3. Transmisja danych

W rozdziale zawarte są podstawowe informacje dotyczące transmisji danych pomiędzy kartą elektroniczną a terminalem (warstwa 1 i 2 modelu OSI).

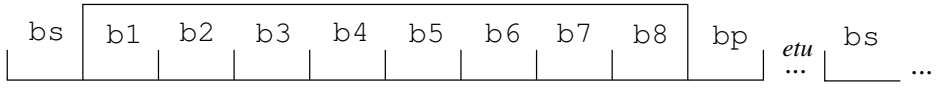
3.1. Warstwa fizyczna

Warstwa fizyczna wymiany danych zdefiniowana została w normach:
— ISO/IEC 7816-3 dla kart stykowych,
— ISO/IEC 10536-3 dla kart bezstykowych.

3.2. Odpowiedź na zerowanie (ATR)

ATR (ang. *answer to reset*) jest odpowiedzią karty na zerowanie. Jest to ciąg znaków zawierający maksymalnie 33 bajty. Rzadkością jest wykorzystanie wszystkich bajtów ATR i najczęściej zawiera się w nim tylko najważniejsze dane, tak aby dostęp do karty był jak najszybszy.

3.2.1. Struktura ATR



Rysunek 10. Struktura znaku (konwencja główna)

Na rysunku 10 przedstawiono strukturę przesyłanego znaku. Transmisja składa się z:
— bs – bit startu,
— b1-b8 – bity danych,
— bp – bit parzystości,
— *etu* – czas ochronny pomiędzy kolejnymi porcjami danych,
— kolejny znak...

Nazwa	Zawartość
TS	znak początkowy
T0	znak formatu
TA1, TB1. TC1, TD1, ...	opcje interfejsu
T1, T2,...,TK	znaki historyczne
TCK	znak kontrolny

Tablica 1. Składniki ATR zgodne z normą ISO/IEC 7816-3

Kolejne składniki ATR (tablica 1) oznaczają:
— TS – bajt obowiązkowy; służy do wyznaczenia szybkości transmisji (terminal wyznacza czas pomiędzy zmianą stanu wysokiego na niski i na tej podstawie oblicza etu (ang. *elementary time unit*) - jest to $\frac{1}{3}$ czasu pomiędzy zboczem rozpoczynającym bit `bs` a zboczem rozpoczynającym trzeci bit danych - jest to postępowanie poprawne ponieważ dla każdej z konwencji pierwsze cztery stany linii I/O są identyczne):
  — '3B' – konwencja główna (ang. *direct convention*); stan wysoki określa logiczną jedynkę; dane przesyłane są od najmłodszego do najstarszego bitu,

— ’3F’ – konwencja odwrócona (ang. *inverse convention*); stan wysoki określa logiczne zero; dane przesyłane są od najstarszego do najmłodszego bitu;

— T0 – drugi obowiązkowy bajt sekwencji ATR; kolejne bity oznaczają które z opcji interfejsu zostaną przesłane oraz określają ilość znaków historycznych (tablica 2);

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
…	…	…	…	x	x	x	x	ilość znaków historycznych (0-15)
…	…	…	1	…	…	…	…	TA1 zostanie nadane
…	…	1	…	…	…	…	…	TB1 zostanie nadane
…	1	…	…	…	…	…	…	TC1 zostanie nadane
1	…	…	…	…	…	…	…	TD1 zostanie nadane

Tablica 2. Znaczenie bitów znaku T0

— opcje interfejsu – definiują wszystkie parametry protokołu komunikacji[3]; łącznie mogą zawierać do 15 bajtów, ale nie są obowiązkowe; wyróżniamy:
  — globalne opcje interfejsu (ang. *global interface characters*) – podstawowe parametry protokołu transmisji,
  — specyficzne opcje interfejsu (ang. *specyfic interface options*) – parametry szczegółowe lub przeznaczone tylko dla określonych protokołów transmisji.

  Znaki wysyłane są w sekwencjach TDi, TAi, TBi, TCi, gdzie i oznacza numer kolejnej sekwencji. Znak TDi zawiera informacje o typie użytego protokołu (zobacz również 3.4) i kolejnych przesyłanych sekwencjach TAi, TBi, TCi (tablica 3). Znaczenie TAi, TBi, TCi

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
…	…	…	…	x	x	x	x	numer protokołu tansmisji (0-15)
…	…	…	1	…	…	…	…	TA$i+1$ będzie nadane
…	…	1	…	…	…	…	…	TB$i+1$ będzie nadane
…	1	…	…	…	…	…	…	TC$i+1$ będzie nadane
1	…	…	…	…	…	…	…	TD$i+1$ będzie nadane

Tablica 3. Znaczenie bitów znaku TDi

zgodnie z ISO/IEC 7816-3 jest następujące:

— TA1 – jest globalną opcją interfejsu; zawiera informacje pozwalające wyznaczyć współczynnik konwersji szybkości zegara F (ang. *clock rate conversion factor*) oraz współczynnik regulacji szybkości transmisji D (ang. *bit rate adjustment factor*); współczynniki te określone są na podstawie wartości FI i DI (tablica 4) - odpowiednie tabele konwersji znajdują się w standardzie ISO/IEC 7816-3; używane są przy określaniu parametrów transmisji,

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
x	x	x	x	…	…	…	…	FI
…	…	…	…	x	x	x	x	DI

Tablica 4. Znaczenie bitów znaku TA1

[3] odkąd w normach zdefiniowano podstawowe parametry opcji interfejsu często nie są one wymagane w sekwencji ATR

— TB1 – to także globalna opcja interfejsu; zawiera opcjonalne wartości napięcia i prądu używanego podczas programowania EPROM określone przez współczynniki II i PI1 (tablica 5); obecnie rzadko używana, ponieważ standardowo używaną pamięcią jest EEPROM - jedynie starsze typy kart ją określają,

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
0	x	x	...	...	...	...	...	współczynnik II
0	...	...	x	x	x	x	x	współczynnik PI1

Tablica 5. Znaczenie bitów znaku TB1

— TC1 – globalna opcja wyznaczająca dodatkowy czas ochronny N (ang. *extra guard time*), czyli czas jaki musi upłynąć pomiędzy kolejnymi znakami odbieranymi przez kartę; bajt TC1 może przyjmować następujące wartości:
  — 0-254 – opóźnienie między znakami wynosi co najmniej TC1+12 etu,
  — 255 i używany jest protokoł T=0 – 12 etu (nie ma żadnego czasu ochronnego gdyż przez dodatkowe 2 etu I/O musi pozostać w stanie wysokim),
  — 255 i używany jest protokoł T=1 – 11 etu;
— TA2 – również globalna opcja określająca dozwolone metody określania wyboru protokołu (PTS, ang. *protocol type selection*, zobacz również 3.3),

b8	...	b6	b5	b4	b3	b2	b1	Znaczenie
0	...	...	...	...	...	...	...	pozwala na zmianę trybu wyboru protokołu (pomiędzy negocjacyjnym a określonym)
1	...	...	...	...	...	...	...	zabrania zmiany trybu wyboru typu protokołu
...	0	0	...	...	...	...	...	RFU
...	...	...	0	...	...	...	...	parametry transferu zdefiniowane *explicite* w opcjach interfejsu
...	...	...	1	...	...	...	...	parametry transferu zdefiniowane *implicite* w opcjach interfejsu
...	...	...	...	x	x	x	x	użycie protokołu T=xxxx

Tablica 6. Znaczenie bitów znaku TA2

— TB2 – ostatnia z globalnych opcji interfejsu; zawiera współczynnik PI2 określający wartość napięcia programowania; obecnie rzadko używana (podobnie jak PI1).

Kolejne pola określają specyficzne opcje interfejsu:

— TC2 – zawiera parametr WI, z którego można wyznaczyć czas oczekiwania podczas pracy (*work waiting time*) – dotyczy protokołu T=0.

Poniższe parametry odnoszą się do protokołu wyspecyfikowanego w TD_{i-1}:

— TAi ($i > 2$) – zawiera maksymalną wartość długości IFSC (ang. *information field size for the card*); wartością domyślną jest 32, ale może ona się wahać od 0 do 254,
— TBi ($i > 2$) – młodsza czwórka bitów zawiera wartość CWI służącą do wyliczenia CWT (ang. *character waiting time*); starsza czwórka bitów zawiera BWI, z którego możemy wyliczyć BWT (ang. *block waiting time*),
— TCi ($i > 2$) – najmłodszy bit określa metodę detekcji błędów (0 - używamy algorytmu LRC (ang. *longitudinal redundancy check*), polegającego na wykonywaniu operacji XOR na kolejnych bajtach; 1 - używamy CRC, norma jednak nie precyzuje wielomianu generacyjnego, przyjmuje się, że jest nim $G(x) = x^{16} + x^{12} + x^5 + 1$ określony w ISO 3309); pozostałe bity są RFU.

— znaki historyczne – długość tej sekcji wynosi maksymalnie 15 bajtów; mogą zawierać wiele różnych informacji np.: używany system operacyjny, identyfikator wydawcy karty, numer seryjny, wersja maski ROM, liczba nieudanych dostępów do karty itp.; sposób kodowania istotnych danych określono w normach ISO/IEC 7816-4 i ISO/IEC 7816-5 zdefiniowano trzy pola wewnątrz bajtów historycznych:
  — wskaźnik kategorii – obowiązkowe o długości 1 bajta (starszy półbajt jest zerem, jedynką lub ósemką; młodszy półbajt zawsze zerem),
  — obiekty danych – opcjonalne o zmiennej długości (zgodne z regułami ASN.1 - format BER-TLV); informacje zawarte to kod kraju, producent układu scalonego, IIN (ang. *issuer identification number*), możliwości karty,
  — wskaźnik statusu – warunkowe o długości 3 bajtów: status życia karty oraz dwa bajty zawierające informację statusową (SW1, SW2);
— znak kontrolny – jest sumą kontrolną XOR bajtów od T0 do ostatniego bajtu przed TCK:
  — jeżeli dozwolony jest tylko protokół T=0 TCK może nie wystąpić,
  — dla protokołu T=1 TCK jest obowiązkowe.

3.2.2. Praktyczne przykłady ATR

Przykładowe ATR otrzymane przy pomocy programu z rozdziału 6.1.1(zobacz wydruk 1) wraz z analizą zamieszczone zostały w tablicach 7 oraz 8.

Element	Wartość	Znaczenie	Konsekwencje
TS	’3B’	konwencja główna	
T0	’86’	Y1=’1000’	zostanie nadane TD1
		K=’0110’=6	6 bajtów historycznych
TD1	’40’	’2’=’0100’	zostanie nadane TC2
		T=0	protokół T=0
TC2	’20’	WI=’20’	czas oczekiwania podczas pracy
T1...T6	’6801010204AC’		specyficzne dane producenta karty

Tablica 7. Analiza ATR karty z serii ActiveCard

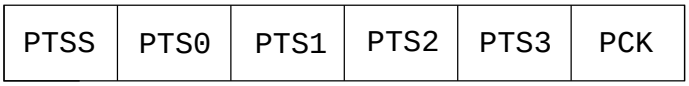
Element	Wartość	Znaczenie	Konsekwencje
TS	’3B’	konwencja główna	
T0	’17’	Y1=’0001’	zostanie nadane TA1
		K=’0111’=7	7 bajtów historycznych
TA1	’11’	FI=’0001’	służy do wyznaczenia współczynnika F
		DI=’0001’	służy do wyznaczenia współczynnika D
T1...T7	’9501010000419F’		specyficzne dane producenta karty

Tablica 8. Analiza ATR przykładowej karty GSM

3.3. Wybór typu protokołu (PTS)

Jeżeli terminal chce zmienić protokół transmisji lub parametry aktualnie stosowanego protokołu (początkowe ustawienia były wymuszone przez zawartość ATR) musi przesłać do karty żądanie PTS (ang. *protocol type selection*). Można je wysyłać jedynie po uzyskaniu ATR od karty.

Tryby wyboru typu protokołu:
— negocjacyjny – wartości współczynników F i D pozostają niezmienione aż do zakończenia procedury PTS,
— określony – wartości F i D ustalone w ATR są obowiązkowe dla procedury PTS.



Rysunek 11. Struktura żądania PTS

Budowa rozkazu PTS jest przedstawiona na rysunku 11. Składa się ona z obowiązkowych bajtów PTSS, PTS0 i PCK. Pozostałe elementy są opcjonalne.

Znaczenie poszczególnych bajtów PTS jest następujące:
— PTSS – znak inicjalizujący żądanie PTS; jego wartość zawsze wynosi ’FF’,
— PTS0 – specyfikuje protokół oraz określa które z opcjonalnych bajtów wystąpią (tablica 9),

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
...	...	...	...		x			numer protokołu transmisji (0-15)
...	...	...	1		...			PTS1 będzie nadane
...	...	1	...		...			PTS2 będzie nadane
...	1	...	...		...			PTS3 będzie nadane
0	...	...	...		...			RFU

Tablica 9. Znaczenie bitów bajtu PTS0

— PTS1 – zawiera nowe wartości współczynników F i D – zakodowane w FI i DI (tablica 10),

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
	x				...			FI
	...				x			DI

Tablica 10. Znaczenie bitów bajtu PTS1

— PTS2 – zawiera nowe wartości parametrów dotyczących między innymi czasu ochrony (tablica 11),
— PTS3 – bajt RFU,
— PCK – znak kontrolny; jest to XOR z poprzednich bajtów.

Struktury żądania i odpowiedzi PTS są identyczne. Jeżeli karta nie obsługuje PTS lub nie jest w stanie zaakceptować wymagań terminala to po wysłaniu żądania terminal nie otrzyma żadnej odpowiedzi. Wtedy zostanie przekroczony czas oczekiwania i karta będzie powtórnie wyzerowana. W przeciwnym wypadku karta odsyła odpowiedź (identyczną jak żądanie).

Wybór typu protokołu możliwy jest także przy zastosowaniu innej procedury. Karta po przesłaniu ATR do terminala jest powtórnie zerowana i przełącza numer obsługiwanego protokołu. Jednak przy większej ilości obsługiwanych przez kartę protokołów procedura ta jest czasochłonna.

b8	b7	b6	b5	b4	b3	b2	b1	Znaczenie
		...				0	0	nie wymagany czas ochronny
		...				0	1	N=255
		...				1	0	dodatkowy czas ochronny (12 etu)
		x				...	...	RFU

Tablica 11. Znaczenie bitów bajtu PTS2

3.4. Protokoły transmisji danych

Po ustaleniu parametrów transmisji pomiędzy kartą a terminalem (ATR, PTS) można przystąpić do wysyłania rozkazów do karty. Protokoły używane do transmisji danych można podzielić na dwie kategorie:

— protokoły stosowane w kartach pamięciowych np.: SLE 4403, I^2C bus (S=8), S=9, S=10,
— protokoły stosowane w kartach inteligentnych np.: T=0, T=1, T=2, T=14 (tablica 12).

Protokół transmisji	Opis
T=0	asynchroniczny, pół-dupleksowy, znakowy (ISO/IEC 7816-3)
T=1	asynchroniczny, pół-dupleksowy, blokowy (ISO/IEC 7816-3)
T=2	asynchroniczny, pełno-dupleksowy, znakowy (ISO/IEC 10536-4)
T=3	pełno-dupleksowy, nie opracowany
T=4	asynchroniczny, pół-dupleksowy, znakowy (rozszerzenie T=0), nie opracowany
T=5...13 i T=15	RFU
T=14	dla zastosowań regionalnych (nie jest w normie ISO)

Tablica 12. Protokoły transmisji (ISO/IEC 7816–3)

3.4.1. Synchroniczna transmisja danych

Protokoły synchronicznej transmisji danych używane są w kartach nie wyposażonych w mikroprocesor. Zwykle są to karty używane w telekomunikacji lub proste elektroniczne portmonetki. Protokoły tego typu wyróżniają się dużą prostotą. Nie ma adresowania logicznego, detekcji i korekcji błędów. W celu maksymalnego uproszczenia budowy terminali i kart protokoły tego typu są mocno powiązane ze sprzętem.

Przykładem protokołu synchronicznej transmisji danych może być protokół używany w kartach telefonicznych opartych na mikroprocesorze Infineon SLE4403. Zarządzanie pamięcią realizowane jest przez cztery funkcje: czytaj, zapisz, wyczyść oraz zwiększ wskaźnik adresu. Karta posiada globalną zmienną wskaźnikową, która pozwala analizować zawartość pamięci bit po bicie.

3.4.2. Protokół T=0

Protokół T=0 jest historycznie pierwszym protokołem dla kart inteligentnych. Został tak zaprojektowany aby zminimalizować użycie pamięci i maksymalnie uprościć transmisję. Jego specyfikacje zawarte są w normach ISO/IEC 7816-3, GSM 11.11 oraz EMV.

Na rysunku 12 przedstawiono budowę komendy w protokole T=0. Składa się ona z nagłówka (bajty CLA, INS, P1, P2, P3) oraz pola danych, którego długość jest wyznaczona przez parametr P3 (zobacz również 3.5.1).

Rysunek 12. Struktura rozkazu w protokole T=0

Jest to protokół zorientowany znakowo. Z każdym bajtem transmitowany jest bit parzystości i w wypadku błędu znak musi być ponownie przesłany.

Wymiana danych pomiędzy terminalem a kartą przebiega w następujący sposób:

— przesłanie do karty nagłówka komendy,
— karta po bezbłędnym otrzymaniu nagłówka przesyła ACK do terminala,
— terminal przesyła dane (o długości określonej w bajcie P3),
— terminal otrzymuje odpowiedź (SW1, SW2) i informacje o ewentualnych przygotowanych danych,
— jeżeli karta przygotowała dane w odpowiedzi uzyskujemy je przy pomocy komendy GET RESPONSE.

Protokół T=0 nie umożliwia przesłania i odebrania danych w jednym cyklu instrukcji oraz przesyłania nagłówka komendy w trybie bezpiecznej wymiany danych[4].

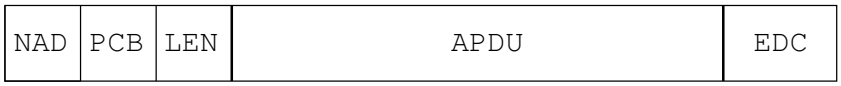
3.4.3. Protokół T=1

T=1 jest asynchronicznym, pół-dupleksowym protokołem transmisji zdefiniowanym w dokumentach ISO/IEC 7816-3 oraz EMV. Należy do warstwy łącza danych modelu OSI. Jest to protokół zorientowany blokowo.

Wyróżniamy trzy rodzaje bloków:

— służące do przekazania danych pomiędzy aplikacjami na karcie i terminalu,
— z danymi kontrolnymi protokołu (bloki systemowe),
— informujące o pozytywnym lub negatywnym odbiorze informacji.

Na rysunku 13 przedstawiono strukturę bloku. Składa się on z:



Rysunek 13. Struktura bloku w protokole T=1

— prologu:
  — bajt NAD (ang. *node address*) – zawiera adres źródłowy i przeznaczenia bloku,
  — bajt PCB (ang. *protocol control byte*) – bajt kontrolny bloku; w zależności od typu przesyłanego bloku może zawierać informacje zmieniające opcje protokołu (np. czas oczekiwania), dotyczące zaistniałych błędów transmisji lub przeznaczone do synchronizacji bloków,
  — bajt LEN (ang. *length*) – zawiera długość pola APDU;
— danych:
  — pole APDU (ang. *application protocol data unit*, 0...254 bajtów) – w bloku tym zawarte są dane przesyłane pomiędzy aplikacjami (warstwa siódma modelu OSI);
— epilogu:
  — pole EDC (ang. *error detection code*, 1 lub 2 bajty) – zawiera dane pozwalające wykryć błędy transmisji.

[4] rozwiązaniem tego problemu jest użycie komendy ENVELOPE

Protokół umożliwia pełną komunikację (przesłanie danych do karty i odpowiedzi) w jednym cyklu instrukcji.

W przypadku błędu podczas komunikacji następuje retransmisja wadliwego bloku. Jeżeli problem się powtórzy będzie wykonana ponowna synchronizacja i retransmisja bloku. Niepowodzenie kończy się zerowaniem karty.

Oddzielenie warstwy danych aplikacji od łącza danych umożliwia wykorzystanie wszystkich możliwości bezpiecznej transmisji danych.

3.4.4. Protokół T=2

T=2 nie jest obecnie protokołem używanym w praktyce. Zgodnie z założeniami ma być to asynchroniczny, pełno-dupleksowy i zorientowany blokowo protokół komunikacyjny. Sposób detekcji błędów, podobnie jak w protokole T=1, ma być oparty na bitach parzystości i EDC.

3.5. APDU

Wymiana danych pomiędzy kartą a terminalem odbywa się przy użyciu APDU (ang. *application protocol data unit*), czyli niepodzielnej struktury danych, która służy do reprezentacji zarówno rozkazu i danych wysyłanych do karty jak i odpowiedzi karty. Specyfikacja APDU z normy ISO/IEC 7816-4 zakłada jego niezależność od protokołu i umiejscawia go w warstwie 7 modelu OSI.

3.5.1. Struktura rozkazu

Rozkaz wysyłany do karty składa się (rysunek 14) z obowiązkowego nagłówka (CLA, INS, P1, P2) i opcjonalnego pola danych (Lc, dane, Le).

CLA	INS	P1	P2	Lc	dane	Le

Rysunek 14. Ogólna struktura rozkazu

CLA reprezentuje klasę rozkazów. Jest używane w celu identyfikacji:
— aplikacji oraz jej specyficznych rozkazów,
— rodzaju bezpiecznej wymiany informacji,
— kanałów logicznych.
Znaczenie poszczególnych bitów bajtu CLA przedstawiono w tabeli 13. Najczęściej występujące klasy rozkazów to:
— ’0x’ – rozkazy zgodne z ISO/IEC 7816-4 (x oznacza cztery młodsze bity CLA),
— ’80’ – elektroniczna portmonetka (EN 1546-3),
— ’8x’ – rozkazy specyficzne dla określonej aplikacji lub firmy produkującej kartę; karty kredytowe zgodne z normą EMV-2,
— ’A0’ – aplikacja GSM (EN 726-3).

INS jest bajtem określającym rozkaz wysyłany do karty. Podstawowe komendy (wraz z wartościami INS) przedstawione są w rozdziale 4.9. Najlepiej jednak skorzystać ze specyfikacji producenta określonej karty lub systemu operacyjnego. Powinna ona zawierać wszystkie rozkazy zaimplementowane na karcie.

P1 i P2 są dodatkowymi parametrami umożliwiającymi przesłanie dodatkowych danych dla wybranej instrukcji INS.

Lc (ang. *command length*) określa długość pola z danymi.

b8…b5	b4	b3	b2	b1	Znaczenie
…	…	…	x	x	numer kanału logicznego
…	0	0	…	…	wyłączone zabezpieczenie danych
…	0	1	…	…	zabezpieczenie danych niekompatybilne z ISO
…	1	0	…	…	zabezpieczenie danych zgodne z ISO (bez uwzględnienia nagłówka)
…	1	1	…	…	zabezpieczenie danych zgodne z ISO (z uwzględnieniem nagłówka)
’0’	…	…	…	…	struktura zgodna z ISO/IEC 7816-4
’8’,’9’	…	…	…	…	struktura zgodna z ISO/IEC 7816-4 komendy charakterystyczne dla aplikacji
’A’	…	…	…	…	struktura zgodna z ISO/IEC 7816-4, dodatkowe komendy (np. z GSM 11.11)
’F’	1	1	1	1	zarezerwowane dla PTS

Tablica 13. Najistotniejsze klasy rozkazów (CLA) – znaczenie bitów

Le (ang. *expected length*) określa spodziewaną długość pola z danymi stanowiącymi odpowiedź karty. Jedynym wyjątkiem jest przesłanie bajtu ’00’ – wtedy terminal oczekuje przesłania największej możliwej porcji danych odpowiedzi przewidzianych dla określonej komendy.

Pola Lc i Le są jednobajtowe. Możliwe jest jednak ich rozszerzenie do trzech bajtów – należy wtedy w pierwszym bajcie przesłać ’00’, a w kolejnych dwóch właściwą długość. Dane mogą mieć wtedy długość do 65536 bajtów. Ta możliwość, mimo iż zgodna ze standardem, nie jest aktualnie implementowana ze względu na ograniczenia ilości pamięci na karcie.

Możemy wyróżnić następujące możliwe struktury rozkazów:
— nagłówek,
— nagłówek, Le,
— nagłówek, Lc, dane,
— nagłówek, Lc, dane, Le.

3.5.2. Struktura odpowiedzi

Odpowiedź karty na przesłany rozkaz składa się z opcjonalnego pola danych i dwóch obowiązkowych bajtów SW1 i SW2 (rysunek 15).

Rysunek 15. Ogólna struktura odpowiedzi

Możliwe wartości bajtu SW1:
— ’61’ – rozkaz zakończony sukcesem,
— ’62’ lub ’63’ – rozkaz zakończony z ostrzeżeniem (kod ’63’ oznacza, że nastąpiły zmiany w pamięci nieulotnej),
— ’64’ lub ’65’ – błąd wykonania rozkazu (kod ’65’ oznacza, że nastąpiły zmiany w pamięci nieulotnej),
— ’67’ lub ’6F’ – błąd sprawdzenia rozkazu,

— pozostałe kody ’6x’ – wykonanie komendy zostało przerwane bez zmiany pamięci nieulotnej.

Kod powrotu o wartości ’9000’ oznacza, że komenda została wykonana w całości i bez żadnych błędów. Jest to przyjęte u praktycznie wszystkich producentów. W celu sprawdzenia znaczenia pozostałych kodów należy skorzystać ze specyfikacji określonego produktu.

Możemy wyróżnić następujące możliwe struktury odpowiedzi:

— SW1 i SW2,
— dane, SW1 i SW2.

3.6. Zabezpieczenie danych

Podczas transmisji danych pomiędzy terminalem a kartą istnieje możliwość ich przechwycenia lub mainpulowania nimi. Potrzebne jest więc zabezpieczenie danych[5] (ang. *secure messaging*). Pod tym pojęciem kryje się szereg mechanizmów opartych m. in. na algorytmach kryptograficznych. Zadaniem zabezpieczenia danych może być zapewnienie ich integralności, niezaprzeczalności i poufności. Istnieje wiele procedur w których spełnia się wybrane z wymienionych wyżej zadań. Jedna z nich, oparta na normach ISO/IEC 7816-4 oraz ISO/IEC 7816-8, polega na zanurzeniu potrzebnych danych w obiektach TLV (ang. *tag, length, value*). Wyróżnia się trzy rodzaje obiektów zdefiniowanych w ramach TLV:

— dla danych przesyłanych tekstem otwartym – mogą zawierać informacje dotyczące obiektów, które będą przesyłane z lub bez użycia zabezpieczenia danych (tagi ’B0’, ’B1’, ’B2’, ’B3’, ’80’, ’81’, ’99’),
— dla danych dotyczących mechanizmów bezpieczeństwa – początkowa wartość dla podpisu cyfrowego, podpis cyfrowy, suma kontrolna, kryptogram:
  — tagi ’8E’, ’9A’, ’BA’, ’9E’ dla obiektów związanych z uwierzytelnieniem,
  — tagi ’82’, ’83’, ’84’, ’85’, ’86’, ’87’ dla obiektów będących kryptogramami;
— dla danych związanych z dodatkowymi funkcjami.

W kolejnych paragrafach przedstawiono kilka z procedur zabezpieczenia danych.

Tryb autentyczności

Tryb autentyczności (ang. *authentic mode procedure*) pozwala zagwarantować integralność APDU podczas transmisji danych. Odbiorca danych (terminal lub karta) może jednoznacznie stwierdzić czy otrzymane informacje nie zostały zmodyfikowane.

Najczęściej do przesyłanych danych jest załączony CCS (ang. *cryptographic checksum*) obliczony z użyciem algorytmu DES. Odbiorca oblicza własny MAC dla otrzymanych danych. Jeżeli jest on zgodny z otrzymanym to karta lub terminal akceptuje dane.

Klucz używany podczas generacji CCS jest najczęściej tymczasowy (generowany np. z numeru karty oraz liczby losowej).

Tryb łączony

Tryb łączony umożliwia nie tylko zapewnienie integralności danych ale także ich poufności poprzez szyfrowanie przesyłanych danych. Przy użyciu jednego z algorytmów kryptograficznych (najczęściej jest to DES) sekcja danych (wraz z obliczonym CCS) komendy APDU zostaje zaszyfrowana.

[5] używamy również określenia „bezpieczna wymiana danych lub informacji”

Licznik rozkazów

Użycie SSC (ang. *send sequence counter*) uniemożliwia dodania lub usunięcia przez trzecią stronę danych przesyłanych pomiędzy terminalem a kartą. Polega na dołączeniu do sekcji danych licznika rozkazu lub na wykonaniu operacji XOR obliczonej z APDU i SSC. Stan początkowy licznika jest ustalany pomiędzy kartą a terminalem.

3.7. Kanały logiczne

Użycie kanałów logicznych pozwala na zrównoleglone wykonywanie do czterech aplikacji na karcie przy użyciu jednego interfejsu komunikacyjnego. Dzięki nim aplikacje mogą być adresowane niezależnie na poziomie logicznym. Stan karty (zweryfikowane kody PIN itp.) są pamiętane oddzielnie dla każdego z używanych kanałów logicznych.

Zastosowania tego mechanizmu, choć w praktyce jeszcze nie spotykane, wydają się być bardzo obiecujące. Przykładem może być połączenie aplikacji GSM z podręcznym notatnikiem lub możliwość przelewania pieniędzy pomiędzy dwiema portmonetkami umieszczonymi na jednej karcie.

Uwagi bibliograficzne

Tematyka poruszona w tym rozdziale została również uwzględniona w [2, 3].

Początkowe części [9,15,21,27] normy ISO/IEC 7816 można odnaleźć w Internecie. W szczególności część trzecia opisuje sposoby transmisji danych. Odpowiednikiem dla kart bezstykowych jest norma [24].

4. Kartowe systemy operacyjne

COS (ang. *chip operating system*) czyli kartowy system operacyjny zarządza aplikacjami na karcie. Jego głównym zadaniem jest zapewnienie bezpieczeństwa przechowywanych danych, realizacja komunikacji pomiędzy kartą a terminalem, zarządzanie systemem plików oraz dodatkowe funkcje takie jak wykonywanie algorytmów kryptograficznych oraz wymiana danych pomiędzy aplikacjami. Obecnie istnieje wiele systemów operacyjnych dla kart, zarówno tych o ogólnej funkcjonalności jak i dedykowanych dla specyficznych aplikacji. W większości przypadków każdy z producentów dostarcza kartę z własnym systemem operacyjnym.

Pierwszym prawdziwym kartowym systemem operacyjnym był, opracowany w roku 1990, *STARCOS*. Pozwalał on na przechowywanie i niezależne zarządzanie kilkoma aplikacjami umieszczonymi na jednej karcie.

Proste aplikacje, które realizowały funkcjonalność podobną do obecnych systemów operacyjnych rozwijały się od 1980 roku. Były przeznaczone głównie dla sieci telefonii komórkowej. Z nich wyewoluował system GSM, który obecnie nie tylko pozwala na podłączenie do systemu GSM, ale również na zarządzanie danymi przechowywanymi w telefonie komórkowym.

Sukcesywnie wzrasta ilość pamięci jaką może zająć system operacyjny na karcie, wzrasta także ich funkcjonalność i niezawodność. COS są implementowane głównie w asemblerze i języku C.

Nowoczesne systemy operacyjne pozwalają nie tylko na zarządzanie danymi na karcie, ale również na umieszczanie aplikacji wykonywanych po stronie karty. Za przykład mogą posłużyć Java Cards, czyli karty z zaimplementowaną maszyną wirtualną języka Java, na których można umieszczać kardlety (aplety języka Java).

Do najważniejszych norm, na których oparte są dzisiejsze systemy operacyjne należy zaliczyć: ISO/IEC 7816, GSM 11.11 oraz EMV.

4.1. Podstawy projektowe

W kartowym systemie operacyjnym, który nie wspiera wykonywania kodu natywnego możemy wyróżnić następujące składniki:

— zarządca I/O oraz interfejs I/O – kontroluje przypływ danych pomiędzy kartą a zewnętrznym interfejsem,
— zarządca bezpiecznej transmisji danych,
— interpretator poleceń oraz zarządca zwracanych danych i kodów powrotu,
— zarządca kanałów logicznych,
— zarządca stanów karty,
— zarządca plików,
— zarządca pamięci EEPROM,
— biblioteka funkcji kryptograficznych.

W normie ISO/IEC 7816-4 zawarto wytyczne projektowe dla systemów operacyjnych (tablica 14). Ich realizacja przez system operacyjny pozwala zaliczyć go do pewnej klasy COS. Średnio system operacyjny zajmuje około 16 kB. Implementacja zarządzania systemem plików (4 rodzaje plików EF, 2 poziomy plików DF) wymaga około 1200 bajtów. Tyle samo potrzeba na zrealizowanie algorytmu DES oraz funkcjonalności protokołu T=1. T=0 to zużycie około 500 bajtów. Zarządca pamięci EEPROM wymaga 300 bajtów, a algorytm CRC 50 bajtów.

Kartowe systemy operacyjne tworzone i testowane są zazwyczaj przy pomocy odpowiednich symulatorów procesorów, na których mają zostać docelowo umieszczone. Pomimo iż kod wygenerowany przez kompilator języka C jest o około 20-40% większy używa się go do implementacji. Dzięki temu system można umieścić na kartach wyposażonych w różne procesory

Klasa	Opis
M	zarządzanie plikami o strukturze przezroczystej oraz stałej liniowej; zaimplementowane komendy: READ BINARY, UPDATE BINARY (bez bezpośredniego wyboru pliku, długość danych do 256 bajtów), READ RECORD, UPDATE RECORD (bez bezpośredniego wyboru pliku), SELECT FILE poprzez FID, VERIFY, INTERNAL AUTHENTICATE (zobacz również 4.9)
N	wytyczne dla klasy M z dodatkową możliwością wyboru pliku DF (zobacz również 4.4.1) przez nazwę
O	zarządzanie plikami o strukturze przezroczystej, stałej liniowej, zmiennej liniowej oraz cyklicznej; zaimplementowane komendy: READ BINARY, UPDATE BINARY (bez bezpośredniego wyboru pliku, długość danych do 256 bajtów), READ RECORD, UPDATE RECORD (bez bezpośredniego wyboru pliku), APPEND RECORD, SELECT FILE, VERIFY, INTERNAL AUTHENTICATE, EXTERNAL AUTHENTICATE, GET CHALLENGE
P	zarządzanie plikami o strukturze przezroczystej; zaimplementowane komendy: READ BINARY, UPDATE BINARY (bez bezpośredniego wyboru pliku, długość danych do 64 bajtów), SELECT FILE z możliwością wyboru DF poprzez nazwę, VERIFY, INTERNAL AUTHENTICATE
Q	zaimplementowane zabezpieczenie przesyłanych danych; zaimplementowane komendy: GET DATA, PUT DATA, SELECT FILE z możliwością wyboru DF poprzez nazwę, VERIFY, INTERNAL AUTHENTICATE, EXTERNAL AUTHENTICATE, GET CHALLENGE

Tablica 14. Klasy kartowych systemów operacyjnych

bez potrzeby przepisywania kodu. Pisząc w języku wyższego poziomu popełnia się także mniej błędów.

Testowanie stworzonego systemu operacyjnego jest bardzo ważnym etapem projektowania. Straty wynikłe z wyprodukowania kart z wadliwym COS mogą być bardzo wysokie, gdyż w tym przypadku nie można umieścić na nich nowej wersji systemu, a jedynie zniszczyć.

4.2. Ładowanie COS na kartę

Jak już wspomniano wcześniej w przypadku wyprodukowania kart z wadliwym systemem operacyjnym umieszczonym w pamięci ROM karta staje się bezużyteczna, gdyż nie można systemu wymienić. Jednym z rozwiązań tego problemu, przynajmniej na etapie testowania kart, jest nagranie w pamięci ROM jednego modułu systemu operacyjnego, który posłuży do umieszczenia jego pozostałej części w pamięci EEPROM. Niestety w ten sposób wzrastają koszty produkcji kart.

4.3. Organizacja pamięci

Karta zazwyczaj zawiera trzy rodzaje pamięci: ROM (zapisywana na etapie produkcji karty), RAM (przechowywane są w niej tymczasowe dane podczas pracy karty, po odcięciu zasilania są tracone), EEPROM (trwała pamięć wielokrotnego zapisu).

W pamięci RAM umieszczane są:

— rejestry,
— stos,
— zmienne,

— tymczasowe dane przechowywane przez funkcje kryptograficzne,
— bufor I/O.

Pamięć EEPROM zawiera:
— informacje producenta,
— elementy systemu operacyjnego,
— wolną przestrzeń przeznaczoną na dane aplikacji.

4.4. Pliki

Dane umieszczane na karcie inteligentnej zorganizowane są w system plików. Umożliwia to regulację dostępu do przechowywanych informacji i ich zabezpieczenie. Zanim wykonamy operację na pliku musi on zostać przez nas wybrany (przy użyciu SELECT FILE). Wynika to z organizacji zapisu pliku w pamięci EEPROM. Wybranie pliku to odczytanie jego nagłówka, który zawiera podstawowe informacje dotyczące obiektu, a w szczególności wskaźnik do danych. W zależności od przyjętych zabezpieczeń dostęp do zapisanych w pliku danych może być możliwy po uprzednim spełnieniu pewnych warunków (np. uwierzytelnieniu). Rozdzielenie nagłówka od danych pliku wpływa także na bezpieczeństwo systemu plików. Nagłówki zapisywane są zazwyczaj w dodatkowo chronionym przez system operacyjny obszarze pamięci.

System plików na kartach inteligentnych jest opisany w normie ISO/IEC 7816-4.

4.4.1. Rodzaje plików

Zasadniczo istnieją dwa rodzaje plików: dedykowane (ang. DF, *dedicated file*, *directory file*) oraz elementarne (ang. EF, *elementary file*). Wśród plików dedykowanych, które odpowiadają katalogom z powszechnie znanych systemów plików takich jak FAT, możemy wyróżnić:
— MF (ang. *master file*) – katalog główny; zawiera wszystkie pozostałe katalogi oraz pliki elementarne; jest automatycznie wybierany po zerowaniu karty, nie może być usunięty,
— DF – katalog zapisany w MF lub w innym katalogu; zawiera inne katalogi i pliki elementarne; nie ma wytycznych określających ilość poziomów DF - jest ona zależna od systemu operacyjnego i dostępnej pamięci.

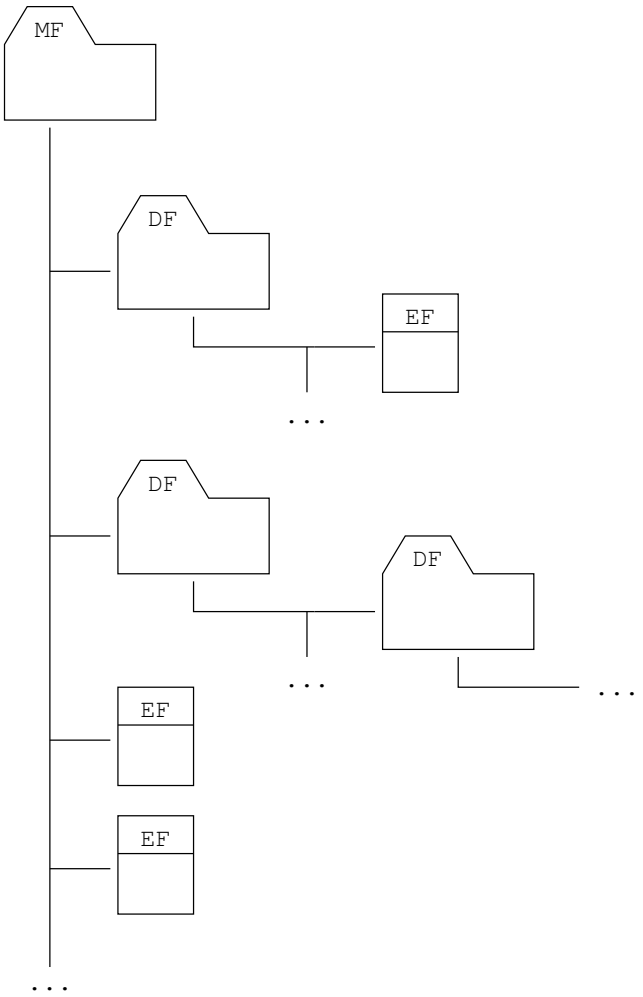
Pliki elementarne mogą być zapisywane w katalogu głównym lub w innym katalogu. Możemy wyróżnić wewnętrzne pliki elementarne (ang. *internal elementary files*) służące do przechowywania danych systemu operacyjnego (nie są one dostępne dla użytkownika) oraz pliki robocze (ang. *working elementary files*) zawierające dane aplikacji.

Na rysunku 16 pokazano przykładowy układ plików na karcie wieloaplikacyjnej.

Aplikacja na karcie posiada zazwyczaj swój katalog, w którym umieszczamy potrzebne pliki. W ten sposób struktura danych na karcie jest przejrzysta. Należy zauważyć, że w wypadku kart, które nie mają możliwości wykonania kodu natywnego, pojęcie „aplikacji na karcie” jest dość nieszczęśliwe i mylące, gdyż nie mamy tu na myśli typowego, wykonywalnego programu. Aplikacja odpowiada tu raczej „danym przechowywanym na użytek aplikacji”, które mogą być umieszczane i pobierane z karty. Pojęcie to weszło już jednak na stałe do użycia i nie zamierzamy go tu zmieniać.

4.4.2. Nazwy plików

Aby skorzystać z określonego pliku przechowywanego na karcie nie musimy pamiętać jego adresu fizycznego. Pliki posiadają swoje identyfikatory, zależne od swojego typu:
— MF – FID (ang. *file identifier*),
— DF – FID, nazwa pliku dedykowanego,
— EF – FID, skrócony FID.



Rysunek 16. Przykład struktury plików na karcie inteligentnej

FID jest dwubajtowym identyfikatorem pliku, który może być używany do wybierania pliku na karcie. Przy nadawaniu FID obowiązują następujące reguły:

— w danym katalogu wszystkie pliki elementarne posiadają różne numery identyfikacyjne,
— katalogi zagnieżdżone nie mogą mieć takich samych numerów FID,
— plik elementarny w katalogu nie może mieć takiego samego identyfikatora jak katalog w którym się znajduje lub katalog zawarty w tym katalogu.

Szereg numerów FID jest zastrzeżonych dla plików o specjalnym przeznaczeniu. Są to między innymi:

— ’3F00’ – katalog główny (MF),
— ’3FFF’ – zarezerwowany do wyboru plików przez ścieżkę,
— ’2F00’ – plik elementarny zawierający identyfikatory aplikacji na karcie (ścieżki do aplikacji),
— ’2F01’ – rozszerzenie ATR (EF),
— ’FFFF’ – zarezerwowany dla przyszłego użycia,
— szereg FID zastrzeżonych w normie EN 726-3 dla plików elementarnych przechowujących dane użytkownika, kody PIN itp.; są to ’0000’, ’0001’, ’0002’, ’0003’, ’0004’, ’0005’, ’0011’, ’0100’, ’2F05’.

Skrócony FID jest wykorzystywany w niektórych komendach do bezpośredniego wyboru pliku. Jest to pięciobitowa liczba, która może przyjmować wartości od 1 do 30. Wartość 0 oznacza aktualnie wybrany plik.

Norma ISO/IEC 7816-4 przewiduje nadawanie nazw plikom dedykowanym. Jest to ciąg bajtów o długości od 1 do 16 bajtów. Na świecie dąży się do identyfikowania aplikacji na karcie poprzez nazwę ich pliku dedykowanego. Norma ISO/IEC 7816-5 opisuje sposób nadawania nazw katalogom. AID (ang. *application identifier*) składa się z obowiązkowego numeru RID (ang. *registered application provider identifier*) o długości 5 bajtów (10 cyfr) i opcjonalnego PIX (ang. *proprietary application identifier extension*) o długości do 11 bajtów. Sposób kodowania numeru RID przedstawiono w tablicy 15.

1	2-4	5-10	Znaczenie
x	...	...	kategoria rejestracji (krajowa lub międzynarodowa)
...	x	...	kod kraju (zgodny z ISO 3166)
...	...	x	przyznany numer aplikacji

Tablica 15. Kodowanie 10 cyfrowego numeru RID

4.4.3. Wybór plików

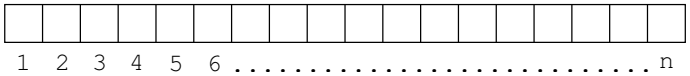
Przed wykonywaniem operacji na pliku należy go wybrać. W zależności od jego typu i możliwości systemu operacyjnego możliwe są następujące sposoby postępowania:

— dla plików dedykowanych – MF może być wybrany z dowolnego miejsca struktury plików; pozostałe katalogi są wybierane poprzez numer FID lub nazwę (AID),
— dla plików elementarnych – wybór poprzez FID przy użyciu komendy SELECT FILE lub bezpośredni wybór poprzez skrócony FID (jako parametr komend operujących na plikach),
— wybór poprzez ścieżkę do obiektu – norma ISO/IEC dopuszcza możliwość wyboru pliku poprzez jego ścieżkę względną (z aktualnego katalogu) lub bezwzględną (z katalogu MF).

4.4.4. Struktura plików elementarnych (EF)

Aby szybko i sprawnie zarządzać danymi przechowywanymi na karcie przewiduje się kilka struktur plików elementarnych. Typ pliku elementarnego definiujemy podczas jego zakładania na karcie. Systemy operacyjne najczęściej oferują następujące struktury:

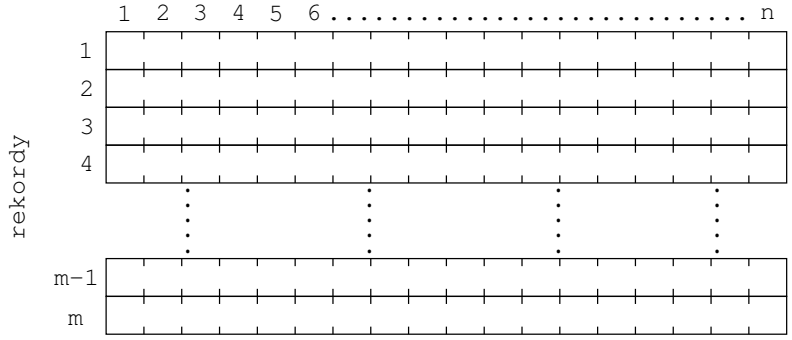
— przezroczysta (rysunek 17) – to najprostsza struktura pliku, bez żadnej organizacji wewnętrznej; żadna z norm nie definiuje maksymalnego rozmiaru takiego pliku (jest on zależny od rodzaju karty); na pliku operujemy przy użyciu instrukcji READ BINARY, WRITE BINARY, UPDATE BINARY podając jako jeden z argumentów offset pliku względem jego początku; ta struktura może być przydatna do przechowywania np. zdjęcia właściciela,



Rysunek 17. Budowa pliku przezroczystego

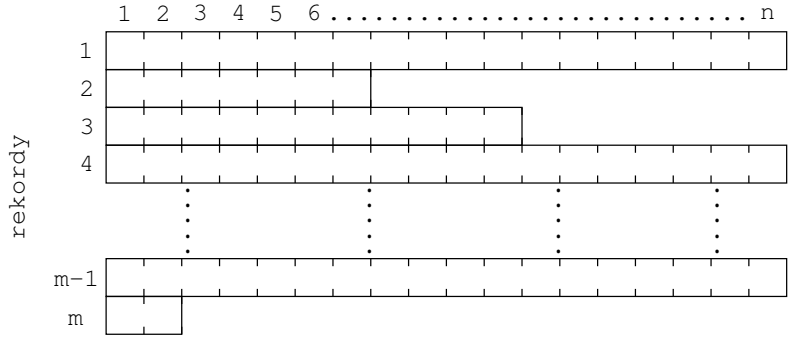
— liniowa stała (rysunek 18) – jest oparta na rekordach o stałej długości definiowanej podczas tworzenia pliku; ilość rekordów oraz ich długość zawiera się w przedziale od 1 do 254 (pierwszy rekord ma numer 1); na pliku operujemy stosując komendy READ RECORD,

WRITE RECORD, UPDATE RECORD podając jako argumenty numer rekordu oraz offset względem początku rekordu; plik tego typu możemy użyć do przechowywania danych książki telefonicznej,



Rysunek 18. Budowa pliku o strukturze stałej liniowej

— liniowa zmienna (rysunek 19) – umożliwia przechowywanie rekordów o różnej długości; pozostałe właściwości są analogiczne jak dla plików o strukturze liniowej stałej; może być wykorzystana do przechowywania danych o właścicielu,

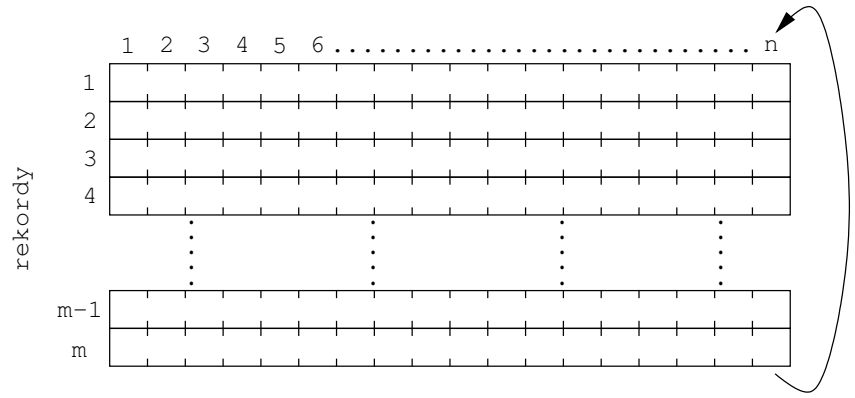


Rysunek 19. Budowa pliku o strukturze zmiennej liniowej

— cykliczna (rysunek 20) – struktura pliku jest analogiczna jak liniowa stała; do rekordów odwołujemy się nie poprzez ich numer a w sposób logiczny: pierwszy, ostatni, następny, poprzedni; system operacyjny pamięta wskazanie do ostatnio używanego rekordu; przykładem zastosowania może być przechowywanie informacji o ostatnich m transakcjach; rekordem następującym po ostatnim jest rekord pierwszy,
— pozostałe struktury: pliki baz danych, wykonywalne, przechowujące obiekty TLV oraz przechowujące informacje o wykonywanych procesach na karcie.

4.4.5. Dostęp do plików

Określone operacje na plikach mogą być możliwe jedynie po spełnieniu pewnych warunków. Dotyczą one stanu w jakim znajduje się karta (np. czy dokonano uwierzytelnienia terminala) oraz zezwolenia na wykonywanie pewnych komend wobec danego pliku:
— dla plików elementarnych: możliwość zapisu, odczytu, blokady, usunięcia pliku,
— dla plików dedykowanych: możliwość tworzenia, usuwania plików.



Rysunek 20. Budowa pliku o strukturze cyklicznej

4.4.6. Atrybuty plików

Atrybuty, które nadają plikom specjalne właściwości, są zależne od stosowanego systemu operacyjnego. Najczęściej możemy spotkać:

— atrybut WORM (ang. *write once, read multiple*) – zapis do pliku z tym atrybutem może być przeprowadzony jednokrotnie, a zachowane dane można odczytywać wielokrotnie; realizowany jest programowo lub sprzętowo; może być użyty do ochrony ważnych danych na karcie,
— przechowywanie wielu kopii pliku – przewidziane dla aplikacji, które często zapisują dane na karcie; służy wydłużeniu żywotności karty,
— wykorzystanie EDC (ang. *error detection code*) – przechowywanie wraz z plikiem jego kodu kontrolnego pozwalającego wykryć (a często skorygować) zaistniałe błędy w pamięci EEPROM; używany np. dla plików przechowujących stan elektronicznej portmonetki,
— atomowość operacji zapisu – określa czy dane powinny być ostatecznie zapisane do pliku tuż po wykonaniu tej operacji,
— wielodostęp – przy wykorzystaniu kanałów logicznych określa czy dany plik jest wielodostępny i w jakich sytuacjach (zapis, odczyt),
— używany interfejs – w kartach posiadających stykowy i bezstykowy interfejs określa przy użyciu którego z nich dany plik jest dostępny.

4.5. Zarządzanie systemem plików

Z każdym plikiem przechowywany jest nagłówek, który zawiera podstawowe informacje o obiekcie:

— FID,
— typ pliku (np. plik elementarny),
— struktura pliku (np. stały, liniowy),
— rozmiar pliku (np. 4 rekordy o długości 10 bajtów),
— prawa dostępu (np. operacja WRITE i UPDATE wymaga podania kodu PIN),
— atrybuty (np. WORM),
— położenie w drzewie katalogów (np. wskazanie na MF).

Nagłówki plików oraz ich wewnętrzne dane przechowywane są w pamięci EEPROM podzielonej na strony. Ze względów bezpieczeństwa w różnych obszarach pamięci umieszcza się pliki przechowujące klucze, kody PIN, czy ogólnie dostępne dane. Dąży się także do oddzielania przestrzeni adresowych umieszczanych aplikacji (czyli katalogów).

Po usunięciu danych z pamięci karty nie jest dokonywana realokacja plików w celu zwiększenia ilości wolnego miejsca. Miejsce dla tworzonych plików jest przydzielane najczęściej zgodnie z ideą algorytmu *best-fit*[6].

Integralność plików jest możliwa poprzez przechowywanie sumy kontrolnej obiektu.

Problemy pojawiają się w sytuacjach awaryjnych. Co stanie się, gdy podczas zapisu do pliku karta zostanie nagle wyciągnięta z czytnika? Dla niektórych aplikacji (np. elektroniczna portmonetka) uszkodzenie danych (np. stan konta) jest niedopuszczalne nawet w tak ekstremalnych przypadkach. Rozwiązaniem jest przechowywanie kopii zapasowych kluczowych danych, a przed, podczas i po zapisie ustawianie specjalnych znaczników informujących o stanie realizacji operacji. W wypadku awarii znaczniki te kontrolowane są podczas startu systemu operacyjnego i dokonywane są odpowiednie kroki mające na celu naprawienie danych i poinformowania systemu zewnętrznego, że ostatnia transakcja zakończyła się z błędem.

Współdzielenie plików na karcie pomiędzy różnymi aplikacjami (np. kodów PIN, kluczy kryptograficznych, informacji o użytkowniku) jest możliwe najczęściej poprzez umieszczanie wspólnych obiektów w katalogu głównym.

4.6. Stan karty

Istnieje wiele sposobów implementacji przechowywania aktualnego stanu karty i dostępu do tych informacji. Mogą one być przechowywane w specjalnym pliku zarządzanym przez system operacyjny o pewnej określonej strukturze. Innym sposobem jest wydzielenie dla tego celu pewnego obszaru pamięci RAM. Oprócz przechowywania stanu karty często w tym obszarze umieszczane są informacje służące do kontroli poprawności aktualnie wykonywanych komend (np. rozmiar wybranego pliku).

4.7. Procesy atomowe

Po przesłaniu komendy do karty i otrzymaniu odpowiedzi mamy pewność, że została ona wykonana. Wiemy, czy zakończyła się sukcesem, czy też błędem. Jednak jeśli podczas wykonywania operacji nastąpią nieoczekiwane okoliczności (jak np. odcięcie zasilania od karty) musimy mieć pewność, że wysłana przez nas komenda została wykonana w całości lub nie została wykonana wogóle (aby nie doszło do modyfikacji tylko części pliku). Jest to kluczowa funkcjonalność w systemach o wysokim ryzyku np. płatniczych.

4.8. Wykonywanie kodu aplikacji po stronie karty

Jednym z impulsów do rozwoju kart i systemów operacyjnych z możliwością ładowania i wykonywania aplikacji był błąd w operacjach zmiennoprzecinkowych jaki wykryto w procesorach Pentium. Nie było możliwości naprawienia tej usterki, gdyż był to błąd sprzętowy a nie programowy. Ogromne koszty jakie poniesiono na wymianę wadliwych procesorów i zmianę technologii produkcji zmobilizowały twórców kartowych systemów operacyjnych. Zadania do jakich ma być przeznaczona karta nie są już realizowane po stronie systemu operacyjnego, na stałe umieszczonego na karcie, a po stronie wgranej na nią aplikacji. Dzięki temu w przypadku wykrycia błędu jego koszty ograniczą się do wymiany aplikacji na kartach. Umożliwia to także rozszerzenie funkcjonalności kart.

Na kartach mogą być umieszczane aplikacje w postaci:

— skompilowanej – kod programu w języku maszynowym mikroprocesora,

[6] algorytm najlepszego dopasowania; polega na wybraniu tego obszaru pamięci, którego wielkość jest najbardziej zbliżona do rozmiaru tworzonego pliku

— interpretowanej – pseudokod lub kod dla maszyny wirtualnej.

Wgranie programu na kartę polega na utworzeniu pliku elementarnego z odpowiednimi atrybutami i jego aktualizację przy pomocy komendy UPDATE BINARY. Początkowa część takiego pliku stanowi tak zwaną liczbę magiczną, która pozwala systemowi operacyjnemu na wykrycie rodzaju pliku wykonywalnego. Przykładowo dla plików z klasami języka Java jest to ciąg bajtów „CAFEBABY”. Wgrany program można wykonać używając komendy EXECUTE.

4.9. Komendy systemu operacyjnego

Komunikacja pomiędzy terminalem a kartą polega na przesyłaniu do karty odpowiednich komend i odbieraniu odpowiedzi. Funkcje jakie są zaimplementowane na karcie zależą od systemu operacyjnego jaki jest w niej umieszczony (czyli od producenta) i powinny być szczegłowo opisane w dokumentacji do danego produktu. Istnieje jednak szereg norm w których zawarte są wytyczne dotyczące popularnych komend (tworzenie systemu plików) lub aplikacji (GSM). Dokument ISO/IEC 7816 opisuje podstawowe funkcje (część 4), bazodanowe (część 7), kryptograficzne (część 8) i dotyczące zarządzania systemem plików (część 9). Normy EMV oraz EN 1546 dotyczą aplikacji płatniczych realizowanych z użyciem kart procesorowych. Funkcje specyficzne dla systemów telekomunikacyjnych (takich jak GSM) przedstawiono w dokumentach GSM 11.11 i EN 726-3.

System operacyjny karty dostarcza zestawu komend które pozwalają użytkować tą kartę zgodnie z jej przeznaczeniem. W częściowo inny zestaw funkcji będzie wyposażona karta dedykowana aplikacji GSM niż płatniczej.

Podstawowymi komendami zdefiniowanymi w normie ISO/IEC 7816-4 są:

— READ BINARY – odczyt pliku,
— WRITE BINARY – zapis do pliku,
— UPDATE BINARY – modyfikacja pliku,
— ERASE BINARY – czyszczenie pliku,
— READ RECORD – odczyt rekordu,
— WRITE RECORD – zapis rekordu,
— APPEND RECORD – dodanie rekordu,
— UPDATE RECORD – modyfikacja rekordu,
— GET DATA – odczyt danych z pliku,
— PUT DATA – zapis danych do pliku,
— SELECT FILE – wybór pliku,
— VERIFY – weryfikacja np. hasła,
— INTERNAL AUTHENTICATE – uwierzytelnienie karty,
— EXTERNAL AUTHENTICATE – uwierzytelnienie terminala,
— GET CHALLENGE – generacja liczby pseudolosowej,
— MANAGE CHANNEL – zarządzanie kanałami logicznymi,
— GET RESPONSE – pobranie odpowiedzi od karty związanej z ostatnio wydanym poleceniem,
— ENVELOPE – umożliwia np. przesyłanie danych, które będą interpretowane jako komenda.

W celu poznania szczegółowego opisu komend najlepiej korzystać z dokumentacji producenta, która dostarczana jest z daną kartą.

4.10. Przykładowe systemy operacyjne

4.10.1. Java Card

Język Java posłużył jako podstawa dla kart *Java Card*. Implementacja uproszczonej maszyny wirtualnej Java pozwala na uruchamianie na karcie apletów kartowych. Szczegółowa specyfikacja tego rodzaju kart została opracowana przez *Sun Microsystem*.

System operacyjny wspierający aplety kartowe dostarcza zestaw komend umożliwiający wgrywanie i zarządzanie aplikacjami (w rzeczywistości jest specjalnym apletem zwanym *Card Manager*).

W celu uproszczenia i ustandaryzowania projektowania apletów zostało przygotowane specjalne API zwane *Java Card Framework*. Jest to zestaw czterech pakietów zawierających szereg przydatnych klas.

Warto wspomnieć o kilku funkcjach, których została pozbawiona maszyna wirtualna na kartach. Najważniejszymi z nich są: brak *garbage collector*[7], brak możliwości stosowania wątków, wielowymiarowych struktur danych, brak typów *double*, *float*, *long*, *char*, brak możliwości klonowania obiektów oraz dynamicznego ładowania klas. Ograniczenia te wynikają z ilości pamięci dostępnej na kartach oraz trudności implementacyjnych.

Przykładami *Java Card* są: *Cyberflex* oraz *GemXpresso*.

Szczegółowy opis projektowania apletów kartowych, wykorzystania API znajduje się w rozdziale 7.2.

4.10.2. Small-OS

Small-OS jest klasycznym systemem operacyjnym dla kart inteligentnych. Oparty jest na normach ISO/IEC 7816-4 oraz GSM 11.11.

Jest to system wieloaplikacyjny typu N (według normy ISO/IEC 7816-4) z dodatkowo zaimplementowanymi komendami: SELECT FILE, READ BINARY, UPDATE BINARY, READ RECORD, UPDATE RECORD, VERIFY, INTERNAL AUTHENTICATE. Posługuje się protokołem transmisji T=1, nie przewiduje wyboru typu protokołu oraz zabezpieczenia danych. Nie ma możliwości wykonywania kodu natywnego. Zaimplementowanym algorytmem kryptograficznym jest DES.

System plików nie umożliwia tworzenia i usuwania plików. Nie ma także zarządzania wolną pamięcią. Pliki robocze mogą mieć strukturę przezroczystą lub stałą liniową. Pliki wewnętrzne (w których przechowywane są np. kody PIN lub klucze) mają strukturę zmienną liniową. Maksymalny rozmiar pliku przezroczystego wynosi 255 bajtów. Można przechować do 2 kodów PIN oraz do 31 kluczy.

Wymagania *Small-OS* to 8 kB pamięci ROM, 1 kB EEPROM, 128 bajtów RAM.

Istnieje symulator *Small-OS* stworzony przez Wolfganga Rankla. *The Smart Card Simulator* wraz z dokumentacją dostępny jest na stronie domowej autora [108].

4.10.3. SOSSE

SOSSE czyli *Simple Operating System for Smartcard Education* to prosty system operacyjny rozwijany przez Matthiasa Bruestle i rozpowszechniany na licencji GNU wraz ze źródłami w języku C. Zawiera implementację komend takich jak:

— WRITE EEPROM – umożliwia bezpośredni zapis do pamięci EEPROM,
— READ EEPROM – odczyt zawartości pamięci EEPROM,
— LED EFFECTS – służy do sterowania diodami świecącymi, które mogą być umieszczone na karcie,

[7] jego zadaniem jest usuwanie niepotrzebnych obiektów z pamięci

— CREATE FILE – utworzenie pliku,
— SELECT FILE – wybranie pliku,
— GET RESPONSE – odebranie danych z karty,
— GET CHALLENGE – generacja liczby losowej przez kartę,
— INTERNAL AUTHENTICATE – uwierzytelnienie,
— komendy zarządzające kodem PIN (zmiana, odblokowanie i weryfikacja kodu PIN).

System przeznaczony jest dla mikrokontrolerów AVR firmy Atmel. Kompilacja może być dokonana pod kontrolą systemu operacyjnego *Linux* oraz *MS Windows*. Do uruchomienia i testowania własnoręcznie dopisanych funkcji nie potrzeba czytnika oraz karty. System można skompilować do postaci umożliwiającej jego emulację na komputerze PC.

Komunikacja oparta jest na protokole T=0.

Szczegółowa dokumentacja zawarta jest w źródłach systemu oraz na stronie internetowej [107]. Dotyczy ona samego systemu operacyjnego, jego kompilacji oraz używanego sprzętu.

4.10.4. MULTOS

MULTOS jest wielozadaniowym systemem operacyjnym przeznaczonym dla kart inteligentnych. Podstawową zaletą tego systemu jest możliwość uruchamiania na jednej karcie wielu aplikacji równocześnie, które pracują zupełnie niezależnie od siebie.

MULTOS jest systemem otwartym co oznacza, że każdy kto pisze aplikacje, produkuje karty, tworzy systemy operacyjne na określone mikroprocesory może wspierać ten system.

MULTOS zapewnia wysoki poziom bezpieczeństwa. Żadne dane zapisane na karcie nie mogą być pobrane bez poprawnej autoryzacji. Dostawcy aplikacji dla *MULTOS* nie muszą być związani ze sobą, nie muszą nawet sobie ufać.

Kolejną ważną cechą tego systemu jest umożliwienie ładowania aplikacji w locie. Oznacza to, że karta wyposażona w *MULTOS* może być wielokrotnie użyta do różnych celów. Jednego dnia może służyć jako karta identyfikacyjna a następnego może zawierać aplikację do ustalania planu podróży pociągiem. Możliwe jest ładowanie aplikacji przez Internet.

Aplikacje dla systemu operacyjnego *MULTOS* są pisane w języku C lub Java. Źródła muszą być skompilowane do postaci MEL (ang. *MULTOS Executable Language*). MEL jest RISC'owym zestawem instrukcji przystosowanym dla *MULTOS*. Przy tworzeniu aplikacji możliwe jest korzystanie z uproszczonych symulatorów.

Aby tworzyć aplikacje dla *MULTOS* trzeba posiadać kartę developerską. Jest to specjalna karta z ustandaryzowaną i uproszczoną metodą ładowania i usuwania certyfikatów.

Ostatnią wersją systemu operacyjnego *MULTOS* jest wersja 4. Organizacja rozwijająca ten system operacyjny - *MOASCO* zapewnia wsteczną zgodność wersji. Projektowana wersja 5 ma obsługiwać dodatkowo karty bezkontaktowe, technologię GSM oraz ECC (ang. *elliptic curve cryptosystem*).

Dzięki temu, że aplikacje dla *MULTOS* uruchamiają się na maszynie wirtualnej zapewniona jest niezależność sprzętowa.

Istnieje standardowy interfejs programisty pomiędzy aplikacjami a systemem operacyjnym (API). Umożliwia to działanie na jednej karcie mikroprocesorowej, z jednym systemem operacyjnym aplikacji dostarczonych od różnych producentów i pisanych przy użyciu zróżnicowanych technologii.

MULTOS wykorzystywany jest głównie w aplikacjach płatniczych.

4.10.5. SetCOS

SetCOS jest przeznaczony do realizacji na karcie elektronicznej głównie systemów związanych z infrastrukturą klucza publicznego (z racji implementacji w systemie symetrycznych

i asymetrycznych algorytmów kryptograficznych). Z pewnymi modyfikacjami system ten wykorzystywany jest również w kartach SIM.

Przewidziano w nim trzy rodzaje plików specjalnych:
— do przechowywania kodów PIN,
— do przechowywania i operacji na kluczach dla algorytmu DES oraz 3DES,
— do przechowywania i operacji na kluczach dla algorytmu RSA.
Karta wyposażona w *SetCOS* ma możliwość uwierzytelnienia użytkownika, terminala oraz zabezpieczenia przesyłanych informacji. Dostęp do danych zawartych w plikach elementarnych jest regulowany za pomocą następujących atrybutów dostępu:
— ALW – dostęp do pliku jest możliwy bez ograniczeń,
— NEV – dane nie mogą zostać nigdy odczytane,
— PIN1 – dostęp możliwy po prawidłowej weryfikacji kodu PIN1,
— PIN2 – dostęp możliwy po prawidłowej weryfikacji kodu PIN2,
— PRO – dostęp możliwy wyłącznie poprzez wymianę zabezpieczonych rozkazów,
— AUT– dostęp możliwy po uwierzytelnieniu terminala względem karty.
W zależności od rodzaju pliku wymienione atrybuty dostępu związane są z różnymi zestawami komend.

Karta posiada zaimplementowane m.in. komendy REHABILITATE i INVALIDATE (aktywacja i dezaktywacja pliku) oraz PERFORM RSA OPERATION (przeznaczona do generacji kluczy oraz korzystania z algorytmu RSA).

4.10.6. Cryptoflex

Cryptoflex jest kartowym systemem operacyjnym firmy Schlumberger (obecnie Axalto). Jest to typowa karta kryptograficzna (algorytmy DES, 3DES oraz RSA). Karta umożliwia generację kluczy asymetrycznych oraz realizację podpisu elektronicznego. Możliwe jest również uwierzytelnienie użytkownika z użyciem kodu PIN oraz terminala z użyciem kluczy algorytmów symetrycznych.

Charakterystyczną cechą karty jest możliwość umieszczenia w systemie operacyjnym karty dodatkowych komend już po stworzeniu elementów elektronicznych karty z maski. Dzięki temu dla ściśle określonych zastosowań możliwe jest dostosowanie systemu operacyjnego bez konieczności tworzenia nowej maski i zmiany technologii.

Karta produkowana jest z różnymi wielkościami pamięci dostępnej dla użytkownika. Ostatnie z wersji tego produktu umożliwiają generację kluczy dla algorytmu RSA o wielkości 2048 bitów.

4.10.7. MPCOS

MPCOS jest przeznaczony do realizacji na karcie elektronicznej portmonetki. Został wyposażony w szereg specjalnych komend m. in.:
— CREDIT – doładowanie elektronicznej portmonetki,
— DEBIT – wydanie gotówki,
— READ BALANCE – odczyt salda elektronicznej portmonetki,
— CANCEL DEBIT – anulowanie ostatniej operacji wydania gotówki,
— SIGN – wygenerowanie podpisu ostatniej transakcji.
W systemie plików został przewidziany specjalny plik reprezentujący elektroniczną portmonetkę. Operacje na nim możliwe są wyłącznie przy użyciu komend płatniczych. W wypadku awarii system jest w stanie odzyskać „gotówkę”.

Na karcie może być przechowywanych kilka elektronicznych portmonetek. Możliwe jest

także ustalenie kwoty, której wydanie nie wymaga podawania kodu PIN. System operacyjny nie przewiduje możliwości usunięcia utworzonych plików.

Szczególną uwagę zwrócono na bezpieczeństwo. Transakcje są zabezpieczone kluczem sesyjnym. Specjalne mechanizmy, takie jak liczniki odwołań do plików, zapobiegają atakom DPA (ang. *differential power analysis*).

4.10.8. GPK

GPK jest kartowym systemem operacyjnym firmy Gemplus. Posiada funkcjonalności kart *MPCOS* rozszerzoną dodatkowo o możliwość korzystania z algorytmu RSA (generacja kluczy, operacje kryptograficzne).

Karta ta, dzięki rozkazowi ERASE, który powoduje wyczyszczenie zawartości karty (komenda ta może zostać nieodwracalnie zablokowana) ułatwia rozwój aplikacji kartowej. Nadaje się również do celów edukacyjnych.

4.10.9. GemClub-Micro

GemClub-Micro jest przykładem systemu dedykowanego. Przeznaczony jest on dla aplikacji lojalnościowych oraz realizacji prostych elektronicznych portmonetek. Zawiera szereg specjalnych typów plików oraz komend.

Oprócz standardowych struktur plików *GemClub-Micro* posiada dodatkowo:

— plik licznika (ang. *counter file*) – przechowuje i zarządza danymi dotyczącymi stanu konta użytkownika (ilości punktów); na karcie można przechowywać do 30 liczników (co można przełożyć na ilość systemów lojalnościowych w jakich może brać udział właściciel),
— plik reguł (ang. *rule file*) – to zestaw makroinstrukcji według których zostają przyznane punkty; na karcie można przechowywać do 16 reguł, a każda z nich może być skojarzona z maksymalnie czterema różnymi licznikami; makroinstrukcja jest logicznym opisem operacji zarówno przyznania jak i odebrania punktów dla danego licznika, pozwala na automatyczne przeliczanie np. wydanej gotówki na punkty,
— plik z kluczem (ang. *secret key file*) – służą do zabezpieczenia wymiany danych i obliczania kryptogramów dla transakcji; każdy z kluczy ma 16 bajtów i jest przechowywany na dwóch 8-bajtowych polach; jeden plik z kluczem może zawierać dokładnie jeden klucz,
— plik z kodem PIN (ang. *secret code file*) – jest używany do zabezpieczenia plików; ma długość 8 bajtów; jeden plik z kodem PIN zawiera dokładnie jeden kod PIN.

Specyficzne komendy to np.: AWARD (przyznanie punktów), REDEEM (odebranie punktów), USE RULE (zastosowanie reguły do określonego licznika). Karta posiada zaimplementowany 3DES.

Uwagi bibliograficzne

Książka [3] zawiera liczne przykłady komend z różnych systemów operacyjnych.

Norma [27] opisuje podstawy projektowe systemów operacyjnych dla kart inteligentnych, rodzaje plików i podstawowe komendy. Operacje bazodanowe (SCQL, ang. *structured card query language*) definiuje [30], kryptograficzne [31], zarządzania kartą [32].

Komendy systemu GSM można odnaleźć w [58]. Analogicznym dokumentem dla EMV jest [33].

System *Cryptoflex* opisany jest szczegółowo w [84, 86]. Informacje o *MULTOS* można odnaleźć w Internecie [101].

5. Czytniki kart

Podstawowym zadaniem czytnika kart inteligentnych jest pośredniczenie pomiędzy kartą a urządzeniem chcącym z niej skorzystać. Czytnik zapewnia także odpowiednie warunki elektryczne dla prawidłowej pracy karty (zasilanie). Dodatkowymi zadaniami może być mechaniczne zabezpieczenie karty na czas transakcji (automatyczne wciągnięcie do czytnika bez możliwości jej wyjęcia przez użytkownika), co ma szczególne znaczenie w np. bankomatach.

5.1. Rodzaje czytników

Ze względu na interfejs komunikacyjny z komputerem lub innym urządzeniem można dokonać następującego podziału:

— port szeregowy RS232 (zasilane zarówno bateryjnie, poprzez port szeregowy oraz poprzez gniazdo PS/2) – są to najpopularniejsze rozwiązania dla komputerów osobistych; obecnie powoli wypierane przez urządzenia komunikujące się z użyciem interfejsu USB, ale nadal atrakcyjne, głównie ze względu na cenę,
— USB (ang. *Universal Serial Bus*) – komunikacja i zasilanie realizowane są poprzez złącze USB,
— zintegrowane z klawiaturą – zasilane z gniazda PS/2; komunikacja odbywa się przez różne interfejsy; dodatkowo klawiatura numeryczna może spełniać funkcję tzw. bezpiecznej klawiatury (np. kod PIN przesyłany jest bezpośrednio do karty, bez udziału oprogramowania),
— ISA (ang. *Industry Standard Architecture*) – stosowane dawniej, czytnik zazwyczaj wbudowany był w obudowę (podobnie jak stacja dysków elastycznych),
— PCMCIA (ang. *Personal Computer Memory Card International Association*) – przeznaczone głównie dla laptopów,
— czytniki terminalowe/bankomatowe – są zazwyczaj zintegrowane z urządzeniem, a do komunikacji z nimi służy specjalny, dedykowany interfejs programisty (zależny od urządzenia, choć funkcjonalnie sprowadzający się do możliwości przesyłania rozkazów do karty),
— eGate – jest rozwiązaniem pozwalającym istotnie uprościć budowę czytnika przerzucając obowiązek obsługi interfejsu na samą kartę; karty zgodne z eGate mają możliwość bezpośredniej komunikacji z wykorzystaniem USB, a czytniki (najczęściej w postaci niewielkich tokenów na karty o rozmiarach karty SIM) są jedynie interfejsem elektrycznym,
— inne rozwiązania – stosowane zazwyczaj w specjalizowanych (np. tokeny do transakcji na elektronicznej portmonetce) lub amatorskich rozwiązaniach.

Mając na uwadze zachowanie czytnika względem urządzenia, które z niego korzysta, można wyróżnić interfejsy:

— pasywne – nie zgłaszają żadnych sygnałów do zewnętrznego urządzenia, potrafią jedynie wykonywać zewnętrzne rozkazy,
— aktywne – zgłaszają sygnały do zewnętrznego urządzenia (np. informujące o włożeniu karty lub jej wyjęciu); przykładem może być architektura PC/SC (zobacz 6.2).

5.2. Phoenix

Phoenix jest prostym interfejsem komunikacyjnym pomiędzy komputerem PC a kartą elektroniczną. Zasilany jest napięciem z zakresu 9 - 30V. Połączenie z komputerem zrealizowane jest z użyciem portu szeregowego z następującymi parametrami komunikacyjnymi: prędkość transmisji - 9600 bit/sek., bit parzystości, transmisja 8 bitowa.

Wykaz potrzebnych elementów jest następujący:

— slot dla kart procesorowych, ośmiopinowy (1 sztuka),

— port RS232 dziewięciopinowy, żeński (1 sztuka),
— układ MAX232 (1 sztuka),
— układ 7805 lub 78L05 (1 sztuka),
— układ 74LS07 lub 7407 (1 sztuka),
— układ 74HC04 (1 sztuka),
— dioda LED zielona (1 sztuka) i czerwona (1 sztuka),
— kondensator 4,7μF/16V lub 2,2μF - 22μF/16 - 63V (5 sztuk),
— kondensator 100nF (1 sztuka),
— kondensator 100pF (1 sztuka),
— kondensator 27pF (2 sztuki),
— opornik 220Ω (2 sztuki),
— opornik 2,2kΩ (2 sztuki),
— opornik 22kΩ (1 sztuka),
— opornik 1MΩ (1 sztuka),
— kryształ 3.579545 MHz (1 sztuka).

Schemat układu i płytkę drukowaną można odnaleźć na stronie internetowej [104].

Zerowanie karty możliwe jest przy odpowiednim sterowaniu sygnałami RTS i CTS. Do programowania układu *Phoenix* można wykorzystać bibliotekę *Smart Card ToolKit* (zobacz 6.7).

Podobnym do *Phoenix* układem jest *Smartmouse*. Różnice występują jedynie przy zerowaniu karty (należy zastosować odwrotny sposób sterowania sygnałami RTS i CTS).

Uwagi bibliograficzne

Różne rodzaje czytników zostały opisane w [3].

6. Obsługa kart w czytnikach

Rozdział ten przedstawia API (ang. *application programming interface*, interfejs programisty) przeznaczone do oprogramowania wykorzystania w aplikacjach czytników oraz terminali[8] obsługujących karty inteligentne (służące do komunikacji komputer-czytnik i czytnik-karta).

W kolejnych sekcjach opisane są:

— interfejsy CT-API (ang. *card terminal application programming interface*) na przykładzie biblioteki dla środowiska linuksowego – zawierają one jedynie podstawowe funkcje do komunikacji z terminalem, są silnie zależne od stosowanego sprzętu (komunikacja z terminalem jak i z samą kartą odbywa się na niskim poziomie - wysyłamy ciągi danych, komend w postaci heksadecymalnej i w podobny sposób odbieramy dane),
— PC/SC (ang. *personal computer/smart card*) – to standard określający sposób obsługi czytników i kart przez system operacyjny oraz interfejs programisty (zobacz 6.2),
— pakiet OpenCT (ang. *Open Card-Terminal*) – został stworzony w celu uproszczenia budowy sterowników dla czytników kart procesorowych oraz sposobu zarządzania nimi; w skład pakietu wchodzi natywny interfejs OpenCT, wtyczki dla PC/SC i CT-API oraz zestaw narzędzi do zarządzania zasobami systemowymi (zobacz 6.3),
— OCF (ang. *Open Card Framework*) – jest zorientowaną obiektowo biblioteką zaimplementowaną w środowisku Java; pozwala rozdzielić architekturę terminala i karty od aplikacji dostarczając standardowe metody komunikacji (zobacz 6.4),
— SATSA (ang. *Security and Trust Services*) – biblioteka w języku Java dla pakietu *Java 2 Platform, Micro Edition* pozwalająca korzystać z zasobów związanych m. in. z kartami procesorowymi (zobacz 6.5),
— SCEZ – niezależna biblioteka w języku C pozwalająca na korzystanie z czytników kart (zobacz 6.6),
— *Smart Card Tool Kit* – biblioteka przeznaczona do obsługi interfejsu Phoenix (zobacz 6.7).

6.1. CT-API

Biblioteka tego typu dostarcza prostego API dla czytników kart.

Po kompilacji pakietu zawierającego interfejs czytnika otrzymamy bibliotekę, którą konsolidujemy z własną aplikacją (rys. 21).

6.1.1. Interfejs programisty

Funkcje do obsługi interfejsu zdefiniowano w `ctapi.h`.

— inicjalizacja urządzenia

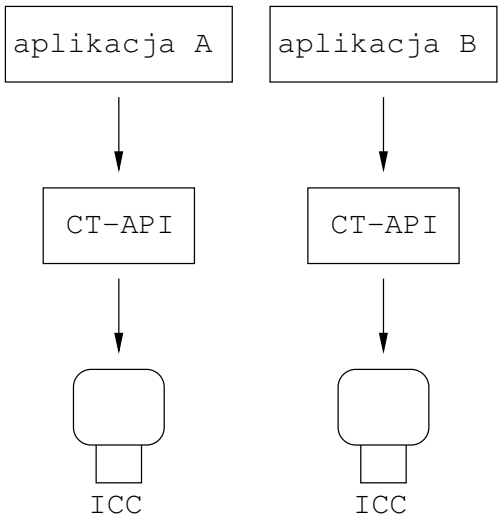
```
char CT_init(unsigned short ctn,
             unsigned short pn);
```

ctn wybrany przez programistę numer czytnika, będzie on używany w późniejszych odwołaniach do interfejsu; dla różnych interfejsów należy oczywiście przyporządkować różne numery

pn numer portu; należy skorzystać z predefiniowanych stałych symbolicznych z pliku nagłówkowego

wartość zwracana funkcja zwraca wartość `OK` gdy zakończy się sukcesem; w przypadku błędu zwraca jedną z wartości reprezentowanych przez stałe symboliczne błędów (plik `ctapi.h`)

[8] w dalszej części publikacji określenia czytnik oraz terminal używane są zamiennie; w wypadku terminali, które najczęściej kojarzą się z urządzeniami o większej funkcjonalności niż sam czytnik, należałoby mówić o wykorzystaniu czytnika terminala



Rysunek 21. Komunikacja w architekturze CT-API

— komunikacja z urządzeniem; funkcja wysyła komendy do karty lub do terminala i zwraca odpowiedź urządzenia

```
char CT_data(unsigned short ctn,
             unsigned char  *dad,
             unsigned char  *sad,
             unsigned short lc,
             unsigned char  *cmd,
             unsigned short *lr,
             unsigned char  *rsp);
```

ctn numer czytnika (wyznaczony przy inicjalizacji urządzenia)

dad wskazuje na adres docelowy

- — komenda wysyłana do terminala – musi się tu znaleźć wartość `CT` ($0x01$)
- — komenda wysyłana do karty – numer identyfikujący otwór w terminalu gdzie włożono kartę (wartości ICCn, $1 \leqslant n \leqslant 14$)
- — po powrocie z funkcji wartość tej zmiennej jest równa wartości `REMOTE_HOST` albo `HOST`

sad wskazuje na adres źródłowy

- — przy wywołaniu funkcji ustawiana na `HOST` albo `REMOTE_HOST` (gdy implementacja CT-API obsługuje zdalne wywołania)
- — po powrocie z funkcji wskazywana wartość jest ustawiona na `CT` albo jedną z ICCn, $1 \leqslant n \leqslant 14$

lc długość polecenia w bajtach

cmd wskaźnik do bufora zawierającego polecenie dla terminala albo karty

lr przed wywołaniem wartość wskazywana przez ten wskaźnik powinna być ustawiona na maksymalną wielkość bufora danych otrzymanych z karty/terminala; po poprawnym wywołaniu funkcji wartość tej zmiennej określa ilość odebranych danych

rsp bufor na dane zwrotne

wartość zwracana funkcja zwraca wartość `OK` gdy zakończy się sukcesem; w przypadku błędu zwraca jedną z wartości reprezentowanych przez stałe symboliczne błędów (plik `ctapi.h`)

— zakończenie komunikacji; zwolnienie wszystkich zasobów przydzielonych dla urządzenia

```
char CT_close(unsigned short ctn);
```

ctn numer czytnika (wyznaczony przy inicjalizacji urządzenia)

wartość zwracana funkcja zwraca wartość `OK` gdy zakończy się sukcesem; w przypadku błędu zwraca jedną z wartości reprezentowanych przez stałe symboliczne błędów (plik `ctapi.h`)

Podstawowe komendy zdefiniowano w pliku `ctbcs.h`.

Na wydruku 1 pokazano przykład użycia funkcji z opisanej biblioteki. Program inicjalizuje kartę, pobiera i drukuje ATR, a następnie dezaktywuje kartę.

```
#include <stdlib.h>
#include "ctapi.h"
#include "ctbcs.h"

// liczba reprezentująca nasz terminal
#define CTERMINAL 1
// makro zwracające określony bit (nbit) z liczby numb
#define BINARY(numb, nbit) ((numb>>nbit & 1) == 0 ? '0' : '1')

// funkcja informująca o błędach
char error(char *fun, char rv)
{
    printf("%s (%i) ", fun, rv);
    switch (rv) {
    case OK:
        printf("OK (Success)\n");
        return OK;
    case ERR_INVALID:
        printf("ERR_INVALID (Invalid Data)\n");
        return ERR_INVALID;
    case ERR_CT:
        printf("ERR_CT (Cardterminal Error)\n");
        return ERR_CT;
    case ERR_TRANS:
        printf("ERR_TRANS (Transmission Error)\n");
        return ERR_TRANS;
    case ERR_MEMORY:
        printf("ERR_MEMORY (Memory Allocation Error)\n");
        return ERR_MEMORY;
    case ERR_HTSI:
        printf("ERR_TRANS (Host Transport Service Interface Error)\n");
        return ERR_HTSI;
    }
    return OK;
}

// funkcja pobiera ATR z terminala o numerze ctn
char get_atr(unsigned short ctn)
{
    char rv;
    unsigned char dad, sad, command[16], response[256];
    unsigned short lenr;
    unsigned short lenc;
    unsigned short p;

    // sprawdzamy czy karta została umieszczona w interfejsie
    command[0] = CTBCS_CLA;
    command[1] = CTBCS_INS_STATUS;
    command[2] = CTBCS_P1_CT_KERNEL;
    command[3] = CTBCS_P2_STATUS_ICC;
    command[4] = 0x00;

    dad = 1;
    sad = 2;
    lenr = 256;

    // wysłanie komendy
```

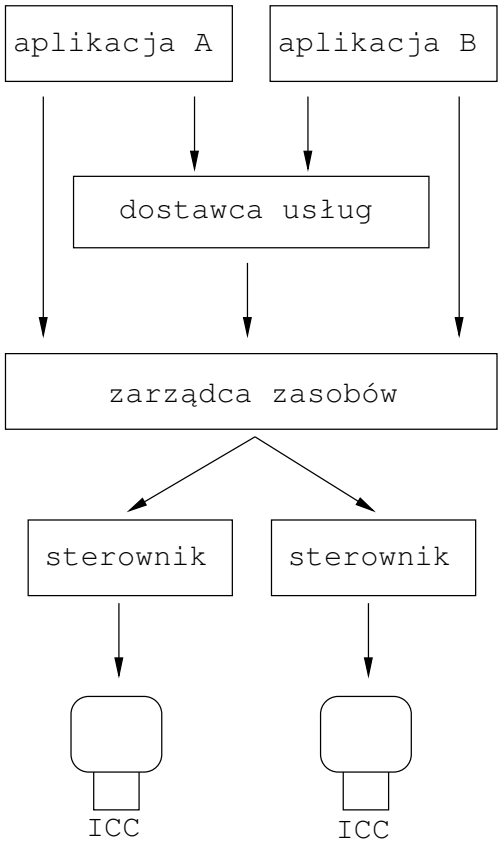
```
    rv = CT_data(ctn, &dad, &sad, 11, command, &lenr, response);

    // wystąpił błąd
    if ((rv != OK) || (response[lenr - 2] != 0x90)) {
        error("CT_data (terminal status)", rv);
        return 0;
    }

    // karta jest załączona
    if (response[0] == CTBCS_DATA_STATUS_CARD_CONNECT) {
        printf("Activiting card...\n");

        // uruchamiamy kartę
        command[0] = CTBCS_CLA;
        command[1] = CTBCS_INS_REQUEST;
        command[2] = CTBCS_P1_INTERFACE1;
        command[3] = CTBCS_P2_REQUEST_GET_ATR;
        command[4] = 0x00;

        dad = 1;
        sad = 2;
        lenr = 256;

        rv = CT_data(ctn, &dad, &sad, 5, command, &lenr, response);

        if ((rv != OK) || (response[lenr - 2] != 0x90)) {
            error("CT_data (activiting card)", rv);
            return 0;
        }

        // wydruk ATR
        for (p = 0; p < lenr - 2; ++p) {
            unsigned short bin;
            printf("[%s%d]  0x%s%X ", p < 10 ? "0" : "", p,
                   response[p] < 16 ? "0" : "", response[p]);
            for (bin = 8; bin > 0; --bin)
                printf("%c", BINARY(response[p], bin - 1));
            printf(" [%c]\n", response[p]);

        }
        return 1;
    } else {
        printf("Please insert a smartcard in the terminal\n");
    }

    return 0;
}

int main(int argc, char *argv[])
{
    char rv;

    // incjowanie połączenia do termnala podłączonego do COM1
    rv = CT_init(CTERMINAL, PORT_COM1);
    if (error("CT_init", rv) != OK)
        return 1;

    get_atr(CTERMINAL);

    // zamykanie połączenia związanego z naszym połączeniem
    rv = CT_close(CTERMINAL);
    if (error("CT_close", rv) != OK)
        return 1;

    return 0;
}
```

Wydruk 1. Wykorzystanie interfejsu CT-API (`CTAPIExample.c`)

6.2. Architektura PC/SC

Architektura PC/SC (ang. *personal computer/smart card*) została wypracowana przez kilkunastu producentów czytników, kart oraz oprogramowania. Jej głównym zadaniem jest uniezależnienie tworzonego kodu od używanej platformy (system operacyjny) oraz sprzętu (czytnik). Na rysunku 22 pokazano elementy architektury PC/SC i zależności między nimi. Aplikacje nie mają bezpośredniego dostępu do zasobów systemowych jakimi są czytniki i umieszczone w nich karty. Zarządca zasobów dostarcza ogólnego interfejsu pozwalającego korzystać z tych urządzeń. Kontroluje on także sposób wykorzystania sprzętu (np. nie pozwala na odwołanie się do karty, która jest aktualnie używana przez inną aplikację). Oprócz bezpośredniego odwoływania się do zarządcy zasobów aplikacje mogą także korzystać z usług oferowanych przez inne interfejsy np. PKCS #11 (zobacz również 10.2.2).

Standard PC/SC zakłada, że wykorzystywane czytniki będą zgłaszały do zarządcy zasobów zdarzenia takie jak włożenie lub wyciągnięcie karty. Dlatego też wykorzystywane urządzenia oraz ich sterowniki muszą być zgodne z tymi zaleceniami.



Rysunek 22. Komunikacja w architekturze PC/SC

W nowych systemach serii *Microsoft Windows* PC/SC jest standardowym sposobem komunikacji z czytnikiem wbudowanym w system operacyjny. Sama implementacja interfejsu jest także najbardziej zaawansowana spośród różnych platform. W wersjach systemu *Microsoft Windows* starszych niż *Microsoft Windows 2000* konieczne jest zainstalowanie zarządcy zasobów.

Implementacją dla środowisk uniksowych jest darmowy pakiet *pcsc-lite*. Dostarcza on większości używanych funkcji i jest zgodny z implementacją firmy *Microsoft*. Działa jako demon systemowy.

Czytniki oraz karty w architekturze PC/SC reprezentowane są poprzez nazwy. Dodatkowo czytniki mogą być połączone w grupy czytników. W systemie *Microsoft Windows* informacje te zawarte są w kluczach rejestru (gałąź HKEY_LOCAL_MACHINE → Software → Microsoft → Cryptography → Calais)

— Readers – czytniki zainstalowane w systemie wraz z informacjami o przynależności do określonej grupy czytników (domyślną grupą jest SCard$DefaultReaders),
— Smartcards – nazwy kart znanych w systemie (rozpoznawane są poprzez ATR); z kartami skojarzona jest biblioteka pozwalająca na ich wysokopoziomową (nie poprzez APDU) obsługę,
— Defaults – w tej gałęzi zdefiniowani są dostawcy usług (biblioteki) dla określonych kart.

Baza danych zawarta w rejestrze systemu może być zarządzana w sposób programowy z użyciem dedykowanych funkcji (pozwalających np. zarejestrować lub usunąć kartę).

Pakiet *pcsc-lite* posiada odpowiedni plik konfiguracyjny, w którym zdefiniowane są nazwy urządzeń oraz odpowiadający im rzeczywisty sprzęt (sterownik, port). Przykład pliku konfiguracyjnego pokazano na wydruku 2. Kolejne linie zawierają nazwę urządzenia, jego fizyczny adres, sterownik urządzenia oraz numer portu, pod który czytnik jest podłączony.

```
FRIENDLYNAME      "Schlumberger Reflex 72 v2"
DEVICENAME        /dev/pcsc/1
LIBPATH           /usr/local/lib/libslbReflex72v2.so
CHANNELID         0x000001

FRIENDLYNAME      "GemPC410"
DEVICENAME        /dev/pcsc/2
LIBPATH           /usr/local/lib/libgp_core.so
CHANNELID         0x0102F8
```

Wydruk 2. Plik konfiguracyjny dla środowiska pcsc-lite (`reader.conf`)

Sterowniki dla czytników USB doładowywane są w trakcie działania demona, po wykryciu podłączenia urządzenia. Każde z urządzeń posiada własny plik konfiguracyjny w formacie XML.

6.2.1. Interfejs programisty (język C)

Do połączenia się z zarządcą zasobów PC/SC i uzyskania kontekstu połączenia przeznaczona jest funkcja:

```
LONG SCardEstablishContext( IN  DWORD           dwScope,
                            IN  LPCVOID         pvReserved1,
                            IN  LPCVOID         pvReserved2,
                            OUT LPSCARDCONTEXT  phContext );
```

dwScope zasięg połączenia (najczęściej `SCARD_SCOPE_SYSTEM` dla połączenia z maszyną lokalną)

pvReserved1 zarezerwowane dla przyszłego użycia (w przypadku połączeń zdalnych może to być adres odległej maszyny)

pvReserved2 zarezerwowane dla przyszłego użycia

phContext kontekst połączenia; zmienna ta będzie używana podczas korzystania z interfejsu zarządcy zasobów

wartość zwracana

— `SCARD_S_SUCCESS` – operacja zakończona powodzeniem
— `SCARD_E_INVALID_VALUE` – nieprawidłowa wartość zmiennej **dwScope**

Do zerwania połączenia z zarządcą zasobów oraz zwolnienia kontekstu służy:

```
LONG SCardReleaseContext( IN SCARDCONTEXT hContext );
```

hContext kontekst połączenia
wartość zwracana
- — SCARD_S_SUCCESS – operacja zakończona powodzeniem
- — SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst

PC/SC w systemie *Microsoft Windows* umożliwia podział czytników na grupy. Odczytanie nazw grup możliwe jest przy użyciu:

```
LONG SCardListReaderGroups ( IN       SCARDCONTEXT hContext,
                             OUT      LPSTR        mszGroups,
                             IN OUT   LPDWORD      pcchGroups );
```

hContext kontekst połączenia
mszGroups bufor, w którym będą umieszczone nazwy grup czytników; każda nazwa oddzielona jest od kolejnej znakiem NULL; bufor kończy się podwójnym znakiem NULL
pcchGroups wielkość bufora
wartość zwracana
- — SCARD_S_SUCCESS – operacja zakończona powodzeniem
- — SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
- — SCARD_E_INSUFFICIENT_BUFFER – zbyt mała wielkość bufora

Do odczytania listy dostępnych czytników przeznaczona jest funkcja:

```
LONG SCardListReaders( IN       SCARDCONTEXT hContext,
                       IN       LPCSTR       mszGroups,
                       OUT      LPSTR        mszReaders,
                       IN OUT   LPDWORD      pcchReaders );
```

hContext kontekst połączenia
mszGroups grupy, do których mają należeć czytniki (w *pcsc-lite* nieużywane)
mszReaders bufor, w którym będą umieszczone nazwy grup czytników; każda nazwa oddzielona jest od kolejnej znakiem NULL; bufor kończy się podwójnym znakiem NULL
pcchReaders wielkość bufora
wartość zwracana
- — SCARD_S_SUCCESS – operacja zakończona powodzeniem
- — SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
- — SCARD_E_INSUFFICIENT_BUFFER – zbyt mała wielkość bufora

Na platformie *Microsoft Windows* zamiast podawania wielkości bufora możliwe jest zastosowanie stałej SCARD_AUTOALLOCATE. Bufor zostanie wtedy automatycznie zaalokowany przez system. Obowiązkiem programisty jest późniejsze zwolnienie pamięci z użyciem funkcji:

```
LONG SCardFreeMemory( IN SCARDCONTEXT hContext,
                      IN LPCVOID     pvMem );
```

hContext kontekst połączenia
pvMem wskaźnik do pamięci, która ma być zwolniona
wartość zwracana SCARD_S_SUCCESS w przypadku powodzenia, inna wartość w przypadku błędu

Kolejna funkcja nawiązuje połączenie z czytnikiem o określonej nazwie. Po nawiązaniu połączenia karta (jeśli jest umieszczona w czytniku) jest zerowana.

```
LONG SCardConnect( IN  SCARDCONTEXT  hContext,
                   IN  LPCSTR        szReader,
                   IN  DWORD         dwShareMode,
                   IN  DWORD         dwPreferredProtocols,
                   OUT LPSCARDHANDLE phCard,
                   OUT LPDWORD       pdwActiveProtocol );
```

hContext kontekst połączenia
szReader nazwa czytnika z którym chcemy nawiązać połączenie
dwShareMode rodzaj połączenia:
- — `SCARD_SHARE_SHARED` – współdzielone
- — `SCARD_SHARE_EXCLUSIVE` – niewspółdzielone

dwPreferredProtocols zalecany protokół komunikacyjny
- — `SCARD_PROTOCOL_T0` – protokół T=0
- — `SCARD_PROTOCOL_T1` – protokół T=1
- — `SCARD_PROTOCOL_RAW` – protokół dla kart pamięciowych

phCard uchwyt do aktualnego połączenia z czytnikiem
pdwActiveProtocol wybrany protokół obsługujący połączenie
wartość zwracana
- — `SCARD_S_SUCCESS` – operacja zakończona powodzeniem
- — `SCARD_E_INVALID_HANDLE` – nieprawidłowy kontekst
- — `SCARD_E_INVALID_VALUE` – nieprawidłowa wartość rodzaju połączenia, protokołu lub nazwa czytnika
- — `SCARD_E_READER_UNAVAILABLE` – nie można uruchomić czytnika
- — `SCARD_E_UNSUPPORTED_FEATURE` – protokół nieobsługiwany
- — `SCARD_E_SHARING_VIOLATION` – czytnik niedostępny (inna aplikacja podłączyła się w trybie niewspółdzielonym)
- — `SCARD_E_NOT_READY` – nie ma możliwości połączenia z portem

W przypadku podłączenia w trybie współdzielonym czasem zachodzi konieczność ponownego połączenia (np. w przypadku gdy inna aplikacja dokonała zerowania karty). Służy do tego funkcja:

```
LONG SCardReconnect( IN  SCARDHANDLE hCard,
                     IN  DWORD       dwShareMode,
                     IN  DWORD       dwPreferredProtocols,
                     IN  DWORD       dwInitialization,
                     OUT LPDWORD     pdwActiveProtocol );
```

hCard uchwyt do połączenia z czytnikiem (uzyskany po wywołaniu funkcji **SCardConnect**)
dwShareMode, dwPreferredProtocols, pdwActiveProtocol zobacz funkcję **SCardConnect**
dwInitialization akcja, jaką należy podjąć tuż po ponownym podłączeniu:
- — `SCARD_LEAVE_CARD` – brak akcji
- — `SCARD_RESET_CARD` – zerowanie karty
- — `SCARD_EJECT_CARD` – wysunięcie karty
- — `SCARD_UNPOWER_CARD` – odłączenie zasilania

wartość zwracana zobacz funkcję **SCardConnect**

Odłączenie od czytnika (zwolnienie uchwytu połączenia) możliwe jest przy użyciu funkcji:

```
LONG SCardDisconnect( IN SCARDHANDLE hCard,
                      IN DWORD       dwDisposition );
```

hCard uchwyt do połączenia z czytnikiem (uzyskany po wywołaniu funkcji **SCardConnect**)
dwDisposition zobacz wartości zmiennej **dwInitialization** dla funkcji **SCardReconnect**
wartość zwracana

- SCARD_S_SUCCESS – operacja zakończona powodzeniem
- SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
- SCARD_E_INVALID_VALUE – nieprawidłowa wartość zmiennej **dwDisposition**

W przypadku konieczności przesłania do karty ciągu niepodzielnych komend należy wykorzystać mechanizm transakcji. Rozpoczęcie transakcji z kartą oznacza dostęp do niej wyłącznie z aplikacji, która go uzyskała, nawet gdy czytnik został udostępniony w trybie dzielonym.

```
LONG SCardBeginTransaction( IN SCARDHANDLE hCard );
```

hCard uchwyt do połączenia z czytnikiem (uzyskany po wywołaniu funkcji **SCardConnect**)
wartość zwracana

- SCARD_S_SUCCESS – operacja zakończona powodzeniem
- SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
- SCARD_E_SHARING_VIOLATION – karta aktualnie niedostępna (inna aplikacja podłączyła się w trybie niewspółdzielonym lub rozpoczęła transakcję z kartą)
- SCARD_E_READER_UNAVAILABLE – czytnik przestał być dostępny

Zakończenie transakcji możliwe jest przy użyciu funkcji:

```
LONG SCardEndTransaction( IN SCARDHANDLE hCard,
                          IN DWORD       dwDisposition );
```

hCard uchwyt do połączenia z czytnikiem (uzyskany po wywołaniu funkcji **SCardConnect**)
dwDisposition zobacz wartości zmiennej **dwInitialization** dla funkcji **SCardReconnect**
wartość zwracana zobacz funkcję **SCardBeginTransaction**

W celu przesłania danych do karty należy wykorzystać funkcję:

```
LONG SCardTransmit( IN      SCARDHANDLE          hCard,
                    IN      LPCSCARD_IO_REQUEST  pioSendPci,
                    IN      LPCBYTE              pbSendBuffer,
                    IN      DWORD                cbSendLength,
                    IN OUT  LPSCARD_IO_REQUEST   pioRecvPci,
                    OUT     LPBYTE               pbRecvBuffer,
                    IN OUT  LPDWORD              pcbRecvLength );
```

hCard uchwyt do połączenia z czytnikiem (uzyskany po wywołaniu funkcji **SCardConnect**)
pioSendPci struktura zawierająca informacje dotyczące wybranego protokołu; najwygodniej użyć zdefiniowanych wartości SCARD_PCI_T0 lub SCARD_PCI_T1
pbSendBuffer bufor z danymi przesyłanymi do karty
cbSendLength wielkość bufora z danymi przesyłanymi do karty
pioRecvPci struktura przeznaczona na informacje dotyczące protokołu
pbRecvBuffer bufor na dane otrzymane od karty
pcbRecvLength wielkość bufora na dane otrzymane od karty
wartość zwracana

- SCARD_S_SUCCESS – operacja zakończona powodzeniem
- SCARD_E_NOT_TRANSACTED – przesłanie komendy APDU zakończyło się niepowodzeniem
- SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
- SCARD_E_READER_UNAVAILABLE – czytnik przestał być dostępny

— SCARD_E_INVALID_VALUE – niepoprawna wartość (np. wielkość bufora)
— SCARD_E_PROTO_MISMATCH – protokół komunikacyjny jest inny niż wybrany
— SCARD_W_RESET_CARD – karta była zerowana (np. przez inną aplikację)
— SCARD_W_REMOVED_CARD – karta została usunięta

Pobranie aktualnych informacji związanych z uchwytem do połączenia z czytnikiem możliwe jest przy użyciu funkcji:

```
LONG SCardStatus( IN      SCARDHANDLE hCard,
                  OUT     LPSTR       szReaderName,
                  IN OUT  LPDWORD     pcchReaderLen,
                  OUT     LPDWORD     pdwState,
                  OUT     LPDWORD     pdwProtocol,
                  OUT     LPBYTE      pbAtr,
                  IN OUT  LPDWORD     pcbAtrLen );
```

hCard uchwyt do połączenia z czytnikiem (uzyskany po wywołaniu funkcji **SCardConnect**)
szReaderName nazwa czytnika związanego z danym uchwytem
pcchReaderLen długość bufora na nazwę czytnika
pdwState zawiera stan w jakim znajduje się karta
— SCARD_ABSENT – brak karty
— SCARD_PRESENT – karta znajduje się w czytniku, ale nie w pozycji umożliwiającej jej wykorzystanie
— SCARD_SWALLOWED – karta znajduje się w czytniku, ale nie jest zasilana
— SCARD_POWERED – karta jest zasilana
— SCARD_NEGOTIABLEMODE – karta była zerowana i oczekuje na wybór typu protokołu (PTS)
— SCARD_SPECIFICMODE – karta była zerowana i wybrany jest pewien protokół komunikacyjny
pdwProtocol używany protokół komunikacyjny
— SCARD_PROTOCOL_T0 – protokół T=0
— SCARD_PROTOCOL_T1 – protokół T=1
— SCARD_PROTOCOL_RAW – protokół dla kart pamięciowych
pbAtr bufor na ATR karty
pcbAtrLen wielkość bufora na ATR karty
wartość zwracana
— SCARD_S_SUCCESS – operacja zakończona powodzeniem
— SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
— SCARD_E_READER_UNAVAILABLE – czytnik przestał być dostępny
— SCARD_E_INSUFFICIENT_BUFFER – zbyt mała wielkość buforów

Oczekiwanie na określone zdarzenie (np. włożenie karty do danego czytnika) możliwe jest do zrealizowania z użyciem funkcji:

```
LONG SCardGetStatusChange( IN      SCARDCONTEXT   hContext,
                           IN      DWORD          dwTimeout,
                           IN OUT
                            LPSCARD_READERSTATE rgReaderStates,
                           IN      DWORD          cReaders );
```

hContext kontekst połączenia
dwTimeout czas oczekiwania, wartość INFINITE oznacza nieskończoność

rgReaderStates tablica struktur definiujących zdarzenia; zajście co najmniej jednego z nich przerywa działanie funkcji

```
typedef struct
{
    LPCTSTR szReader;
    LPVOID  pvUserData;
    DWORD   dwCurrentState;
    DWORD   dwEventState;
    DWORD   cbAtr;
    BYTE    rgbAtr[36];
} SCARD_READERSTATE;
typedef SCARD_READERSTATE *PSCARD_READERSTATE,
                          **LPSCARD_READERSTATE;
```

kolejne pola oznaczają: nazwę czytnika, dane zdefiniowane przez użytkownika, aktualny stan, oczekiwany stan, długość bufora oraz bufor na ATR; wartości zdarzeń są następujące:
- SCARD_STATE_UNAWARE – aplikacja nie zna aktualnego stanu i chiałaby go poznać
- SCARD_STATE_IGNORE – czytnik powinien być ignorowany
- SCARD_STATE_CHANGED – zmiana stanu czytnika
- SCARD_STATE_UNKNOWN – czytnik nie został rozpoznany
- SCARD_STATE_UNAVAIBLE – informacje o stanie czytnika są niedostępne
- SCARD_STATE_EMPTY – brak karty w czytniku
- SCARD_STATE_PRESENT – karta obecna w czytniku
- SCARD_STATE_ATRMATCH – karta w czytniku ma określone ATR
- SCARD_STATE_EXCLUSIVE – czytnik jest używany przez inną aplikację w trybie niewspółdzielonym
- SCARD_STATE_INUSE – karta jest używana przez kilka aplikacji
- SCARD_STATE_MUTE – karta w czytniku nie odpowiada

cReaders ilość struktur w tablicy **rgReaderStates**
wartość zwracana
- SCARD_S_SUCCESS – operacja zakończona powodzeniem
- SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst
- SCARD_E_INVALID_VALUE – nieprawidłowe wartości zmiennych
- SCARD_E_READER_UNAVAILABLE – czytnik przestał być dostępny

W celu przerwania wszystkich oczekujących na zmianę statusu czytnika lub karty funkcji należy zastosować:

```
LONG SCardCancel( IN SCARDCONTEXT hContext );
```

hContext kontekst połączenia
wartość zwracana
- SCARD_S_SUCCESS – operacja zakończona powodzeniem
- SCARD_E_INVALID_HANDLE – nieprawidłowy kontekst

Na wydruku 3 zamieszczono pełny przykład komunikacji z użyciem funkcji PC/SC.

```
// dołączenie niezbędnych plików nagłówkowych
#include <stdio.h>
#include <stdlib.h>

// prototypy funkcji PC/SC
#include <winscard.h>

// wielkość bufora na nazwę czytnika
```

```
#define MAX_READER_NAME_SIZE 40

// w systemie Windows nie ma definicji maksymalnego rozmiaru ATR
#ifndef MAX_ATR_SIZE
#define MAX_ATR_SIZE 33
#endif

int main(int argc, char **argv)
{
    // kontekst połączenia do zarządcy zasobów
    SCARDCONTEXT hContext;
    // uchwyt połączenia do czytnika
    SCARDHANDLE hCard;
    // stan czytnika
    SCARD_READERSTATE_A rgReaderStates[1];
    // pomocnicze zmienne (długości buforów, stan czytnika, protokół)
    DWORD dwReaderLen, dwState, dwProt, dwAtrLen;
    DWORD dwPref, dwReaders, dwRespLen;
    // bufor na nazwę czytnika
    LPSTR pcReaders;
    // bufor na listę czytników
    LPSTR mszReaders;
    // bufor na ATR
    BYTE pbAtr[MAX_ATR_SIZE];
    // bufor na odpowiedź karty
    BYTE pbResp[10];
    // pomocnicze zmienne
    LPCSTR mszGroups;
    LONG rv;
    int i, p, iReader;
    int iReaders[16];

    // komenda GET CHALLENGE
    BYTE GET_CHALLENGE[] = {0x00, 0x84, 0x00, 0x00, 0x08};

    // nawiązanie komunikacji z lokalnym zarządcą zasobów
    printf("SCardEstablishContext : ");
    rv = SCardEstablishContext(SCARD_SCOPE_SYSTEM, NULL, NULL, &hContext);

    if (rv != SCARD_S_SUCCESS)
    {
        printf("failed\n");
        return -1;
    }
    else printf("success\n");

    // pobranie wielkości ciągu, jaki będzie potrzebny na listę
    // dostępnych czytników w systemie
    mszGroups = 0;
    printf("SCardListReaders : ");
    rv = SCardListReaders(hContext, mszGroups, 0, &dwReaders);

    if (rv != SCARD_S_SUCCESS)
    {
        SCardReleaseContext(hContext);
        printf("failed\n");
        return -1;
    }
    else printf("success\n");

    // alokacja pamięci
    mszReaders = (LPSTR) malloc(sizeof(char) * dwReaders);

    // pobranie listy czytników
    printf("SCardListReaders : ");
    rv = SCardListReaders(hContext, mszGroups, mszReaders, &dwReaders);

    if (rv != SCARD_S_SUCCESS)
    {
```

```
        SCardReleaseContext(hContext);
        free(mszReaders);
        printf("failed\n");
        return -1;
    }
    else printf("success\n");

    // wydruk listy znalezionych czytników
    p = 0;
    for (i = 0; i < dwReaders - 1; ++i)
    {
        iReaders[++p]=i;
        printf("Reader %02d: %s\n", p, &mszReaders[i]);
        // przesunięcie bufora do kolejnej nazwy czytnika
        while (mszReaders[++i] != '\0') ;
    }

    // wybór czytnika do dalszych operacji
    do
    {
        printf("Select reader : ");
        scanf("%d", &iReader);
    }
    while (iReader > p || iReader <= 0);

    // wypełnienie struktury stanu czytnika (nazwa czytnika i jego stan)
    rgReaderStates[0].szReader = &mszReaders[iReaders[iReader]];
    rgReaderStates[0].dwCurrentState = SCARD_STATE_EMPTY;

    printf("Insert card...\n");

    // oczekiwanie na zmianę stanu czytnika (włożenie karty)
    printf("SCardGetStatusChange : ");
    rv = SCardGetStatusChange(hContext, INFINITE, rgReaderStates, 1);

    printf("[%02d]\n", rv);

    // nawiązanie połączenia z czytnikiem
    printf("SCardConnect : ");
    rv = SCardConnect(hContext, &mszReaders[iReaders[iReader]],
        SCARD_SHARE_SHARED, SCARD_PROTOCOL_T0 | SCARD_PROTOCOL_T1,
        &hCard, &dwPref);

    if (rv != SCARD_S_SUCCESS)
    {
        SCardReleaseContext(hContext);
        free(mszReaders);
        printf("failed\n");
        return -1;
    }
    else printf("success\n");

    // sprawdzenie stanu karty w czytniku
    printf("SCardStatus : ");
    dwReaderLen = MAX_READER_NAME_SIZE;
    pcReaders = (LPSTR) malloc(sizeof(char) * MAX_READER_NAME_SIZE);

    rv = SCardStatus(hCard, pcReaders, &dwReaderLen, &dwState,
            &dwProt, pbAtr, &dwAtrLen);

    if (rv != SCARD_S_SUCCESS)
    {
        SCardDisconnect(hCard, SCARD_RESET_CARD);
        SCardReleaseContext(hContext);
        free(mszReaders);
        free(pcReaders);
        printf("failed\n");
        return -1;
    }
```

```
        else printf("success\n");

        // wydruk pobranych informacji
        printf("Reader name : %s\n", pcReaders);
        printf("Reader state : %lx\n", dwState);
        printf("Reader protocol : %lx\n", dwProt - 1);
        printf("Reader ATR size : %d\n", dwAtrLen);
        printf("Reader ATR value : ");

        // wydruk ATR
        for (i = 0; i < dwAtrLen; i++)
        {
            printf("%02X ", pbAtr[i]);
        }
        printf("\n");
        free(pcReaders);

        // rozpoczęcie transakcji z kartą
        printf("SCardBeginTransaction : ");
        rv = SCardBeginTransaction(hCard);
        if (rv != SCARD_S_SUCCESS)
        {
            SCardDisconnect(hCard, SCARD_RESET_CARD);
            SCardReleaseContext(hContext);
            printf("failed\n");
            free(mszReaders);
            return -1;
        }
        else printf("success\n");

        // przesłanie do karty komendy GET CHALLENGE
        printf("SCardTransmit : ");
        dwRespLen = 10;
        rv = SCardTransmit(hCard, SCARD_PCI_T0, GET_CHALLENGE,
                           5, NULL, pbResp, &dwRespLen);

        if (rv != SCARD_S_SUCCESS)
        {
            SCardDisconnect(hCard, SCARD_RESET_CARD);
            SCardReleaseContext(hContext);
            printf("failed\n");
            free(mszReaders);
            return -1;
        }
        else printf("success\n");
        printf("Response APDU : ");

        // wydruk odpowiedzi karty
        for (i = 0; i < dwRespLen; i++)
        {
            printf("%02X ", pbResp[i]);
        }
        printf("\n");

        // zakończenie transakcji z kartą
        printf("SCardEndTransaction : ");
        rv = SCardEndTransaction(hCard, SCARD_LEAVE_CARD);
        if (rv != SCARD_S_SUCCESS)
        {
            SCardDisconnect(hCard, SCARD_RESET_CARD);
            SCardReleaseContext(hContext);
            printf("failed\n");
            free(mszReaders);
            return -1;
        }
        else printf("success\n");

        // odłączenie od czytnika
        printf("SCardDisconnect : ");
```

```
        rv = SCardDisconnect(hCard, SCARD_UNPOWER_CARD);

        if (rv != SCARD_S_SUCCESS)
        {
220         SCardReleaseContext(hContext);
            printf("failed\n");
            free(mszReaders);
            return -1;
        }
225     else printf("success\n");

        // odłączenie od zarządcy zasobów PC/SC
        printf("SCardReleaseContext : ");
        rv = SCardReleaseContext(hContext);
230
        if (rv != SCARD_S_SUCCESS)
        {
            printf("failed\n");
            free(mszReaders);
235         return -1;
        }
        else printf("success\n");

        // zwolnienie pamięci
240     free(mszReaders);

        return 0;
    }
```

Wydruk 3. Aplikacja wykorzystująca interfejs PC/SC - język C (`PCSCExample.c`)

6.2.2. Interfejs programisty (język Perl)

Pakiet *pcsc-perl*[9] dostarcza interfejsu do komunikacji z zarządcą zasobów oraz czytnikami z poziomu skryptów w języku Perl.

Klasa **Chipcard::PCSC** umożliwia dostęp do zarządcy zasobów. Obiekt tej klasy można utworzyć przy użyciu jednego z trzech konstruktorów:

```
$hContext=new Chipcard::PCSC($scope, $remote_host);
$hContext=new Chipcard::PCSC($scope);
$hContext=new Chipcard::PCSC();
```

$scope rodzaj połączenia do zarządcy zasobów:
- — `$Chipcard::PCSC::SCARD_SCOPE_SYSTEM` – połączenie lokalne
- — `$Chipcard::PCSC::SCARD_SCOPE_GLOBAL` – połączenie zdalne

$remote_host adres maszyny (w przypadku użycia połączenia zdalnego), wartość 0 oznacza adres maszyny lokalnej

wartość zwracana w przypadku błędu zmienna `$hContext` przyjmuje wartość `undef` (rodzaj błędu można odczytać wykorzystując zmienną `$Chipcard::PCSC::errno`); przypadek przeciwny oznacza, że zmienna `$hContext` jest poprawnym uchwytem do zarządcy zasobów

Użycie drugiego i trzeciego rodzaju konstruktora odpowiada odpowiednio użyciu:

```
$hContext=new Chipcard::PCSC($scope, 0);
$hContext=new Chipcard::PCSC(
    $Chipcard::PCSC::SCARD_SCOPE_SYSTEM, 0);
```

Uzyskany kontekst połączenia służy do wywoływania metod dostarczanych przez klasę. Aby uzyskać listę czytników kart dostępnych w systemie należy skorzystać z:

[9] `http://ludovic.rousseau.free.fr/softwares/pcsc-perl/`

```
$hContext->hContext->ListReaders($group);
```

$group grupa, do której należą czytniki kart (wartość 0 oznacza wszystkie grupy)
wartość zwracana zwracana jest tablica czytników; w przypadku błędu ma ona wartość `undef`

Oczekiwanie na zajście określonego zdarzenia możliwe jest z użyciem metod:

```
$hContext->GetStatusChange(\@readers_states, $timeout);
```

\@readers_states zbiór wartości w postaci słowników zawierających następujące klucze:
- `reader_name` – nazwa czytnika
- `current_state` – aktualny stan czytnika
- `event_state` – oczekiwany stan czytnika
- `ATR` – ATR karty

$timeout maksymalny czas oczekiwania; pominięcie tej wartości spowoduje ustawienie czasu na wartość 0xFFFFFFFF (nieskończoność)

Czas oczekiwania na odpowiedź od zarządcy zasobów (dotyczy w szczególności połączeń zdalnych) można ustalić korzystając z metody:

```
$hContext->SetTimeout($timeout);
```

$timeout czas oczekiwania (w sekundach); pominięcie tej wartości spowoduje ustalenie czasu oczekiwania na 5 sekund

Pomocnicze metody, pozwalające na konwersję ciągów znaków z formatu tekstowego na binarny i odwrotnie to:

```
Chipcard::PCSC::ascii_to_array($apdu);
Chipcard::PCSC::array_to_ascii($apdu_resp);
```

$apdu, $apdu_resp ciągi znaków w określonych formatach
wartość zwracana skonwertowany ciąg znaków

Metody te używane są w szczególności podczas przesyłania komend APDU i odczytywaniu odpowiedzi karty.

Metod do komunikacji z wybranym czytnikiem dostarcza klasa **Chipcard::PCSC::Card**. Konstruktory tego obiektu określone są w następujący sposób:

```
$hCard=new Chipcard::PCSC::Card($hContext);
$hCard=new Chipcard::PCSC::Card($hContext,
       $reader_name, $share_mode, $prefered_protocol);
$hCard=new Chipcard::PCSC::Card($hContext, $reader_name,
       $share_mode);
$hCard=new Chipcard::PCSC::Card($hContext, $reader_name);
```

$hContext uchwyt do zarządcy zasobów
$reader_name nazwa czytnika
$share_mode rodzaj połączenia:
- `$Chipcard::PCSC::SCARD_SHARE_EXCLUSIVE` – aplikacja nie współdzieli czytnika
- `$Chipcard::PCSC::SCARD_SHARE_SHARED` – do używanego czytnika mają dostęp inne aplikacje

$prefered_protocol protokół jaki powinien być użyty (jeśli istnieje taka możliwość):
- `$Chipcard::PCSC::SCARD_PROTOCOL_T0` – protokół T=0

— `$Chipcard::PCSC::SCARD_PROTOCOL_T1` – protokół T=1
— `$Chipcard::PCSC::SCARD_PROTOCOL_RAW` – protokół dla kart pamięciowych

wartość zwracana w przypadku błędu zmienna `$hCard` przyjmuje wartość `undef` (rodzaj błędu można odczytać korzystając ze zmiennej `$Chipcard::PCSC::errno`), w przeciwnym przypadku jest poprawnym uchwytem do czytnika kart

Pierwszy z konstruktorów tworzy obiekt nie powiązany z żadnym z czytników kart. Użycie dwóch ostatnich konstruktorów jest równoważne wywołaniom:

```
$hCard=new Chipcard::PCSC::Card($hContext, $reader_name,
       $share_mode, $Chipcard::PCSC::SCARD_PROTOCOL_T0 |
                    $Chipcard::PCSC::SCARD_PROTOCOL_T1);
$hCard=new Chipcard::PCSC::Card($hContext, $reader_name,
       $Chipcard::PCSC::SCARD_SHARE_EXCLUSIVE,
       $Chipcard::PCSC::SCARD_PROTOCOL_T0 |
       $Chipcard::PCSC::SCARD_PROTOCOL_T1);
```

Do nawiązania połączenia z czytnikiem przeznaczona jest metoda:

```
$hCard->Connect($reader_name, $share_mode, $prefered_protocol);
$hCard->Connect($reader_name, $share_mode);
$hCard->Connect($reader_name);
```

Wartości określonych parametrów są analogiczne do wartości podawanych podczas użycia konstruktora. Użycie dwóch ostatnich metod jest równoważne wywołaniom:

```
$hCard->Connect($reader_name, $share_mode,
        $Chipcard::PCSC::SCARD_PROTOCOL_T0 |
        $Chipcard::PCSC::SCARD_PROTOCOL_T1);
$hCard->Connect($reader_name,
        $Chipcard::PCSC::SCARD_SHARE_EXCLUSIVE,
        $Chipcard::PCSC::SCARD_PROTOCOL_T0 |
        $Chipcard::PCSC::SCARD_PROTOCOL_T1);
```

Kolejna metoda umożliwia ponowne otwarcie połączenia lub zmiana jego ustawień:

```
$hCard->Reconnect($share_mode, $prefered_protocol,
        $initialization);
$hCard->Reconnect($share_mode, $prefered_protocol);
$hCard->Reconnect($share_mode);
$hCard->Reconnect();
```

$share_mode, $prefered_protocol zobacz opisy parametrów konstruktora
$initialization akcja podejmowana przy połączeniu

— `$Chipcard::PCSC::SCARD_LEAVE_CARD` – brak akcji
— `$Chipcard::PCSC::SCARD_RESET_CARD` – zerowanie karty
— `$Chipcard::PCSC::SCARD_UNPOWER_CARD` – wyłączenie zasilania podczas zamknięcia połączenia
— `$Chipcard::PCSC::SCARD_EJECT_CARD` – wysunięcie karty

Trzy ostatnie przypadki użycia metody są równoważne wywołaniom:

```
$hCard->Reconnect($share_mode, $prefered_protocol,
        $Chipcard::PCSC::SCARD_LEAVE_CARD);
$hCard->Reconnect($share_mode,
        $Chipcard::PCSC::SCARD_PROTOCOL_T0 |
        $Chipcard::PCSC::SCARD_PROTOCOL_T1,
        $Chipcard::PCSC::SCARD_LEAVE_CARD);
$hCard->Reconnect($Chipcard::PCSC::SCARD_SHARE_EXCLUSIVE,
```

```
        $Chipcard::PCSC::SCARD_PROTOCOL_T0 |
        $Chipcard::PCSC::SCARD_PROTOCOL_T1,
        $Chipcard::PCSC::SCARD_LEAVE_CARD);
```

Zamknięcie połączenia realizowane jest z użyciem metody:

```
$hCard->Disconnect($initialization);
$hCard->Disconnect();
```

Parametr `$initialization` przyjmuje wartości analogiczne jak przy poprzedniej metodzie. Wywołanie tej metody bez parametrów jest równoważne wywołaniu z parametrem o wartości `$Chipcard::PCSC::SCARD_EJECT_CARD`.

Aktualny stan połączenia można sprawdzić używając metody:

```
$hCard->Status();
```

wartość zwracana tablica z następującymi wartościami:

- `$reader_name` – nazwa czytnika
- `$reader_state` – aktualny stan czytnika
- `$protocol` – oczekiwany stan czytnika
- `\@atr` – ATR karty

Rozpoczęcie transakcji z kartą uzyskujemy przy użyciu:

```
$hCard->BeginTransaction();
```

wartość zwracana jedna z wartości:

- `TRUE` – transakcja rozpoczęta
- `FALSE` – wystąpił błąd podczas rozpoczęcia transakcji

Na przesłanie danych do karty i uzyskanie odpowiedzi pozwala metoda:

```
$resp=$hCard->Transmit(\@data);
```

\@data dane przesyłane do karty
wartość zwracana odpowiedź od karty

W tym celu można również użyć metody, która pozwala na kontrolę odpowiedzi karty:

```
($sw, $resp)=$hCard->TransmitWithCheck($apdu, $sw_expected,
                    $debug);
```

$data dane przesyłane do karty
$sw_expected oczekiwane bajty statusu
$debug parametr opcjonalny; wartość 1 spowoduje wydruk na ekranie danych przesyłanych i otrzymanych z karty
wartość zwracana odpowiedź od karty; w przypadku niezgodności z przewidywanymi bajtami statusu zwracana jest wartość `undef`

Zakończenie transakcji z kartą możliwe jest dzięki metodzie:

```
$hCard->EndTransaction($disposition);
$hCard->EndTransaction();
```

$disposition akcja podejmowana podczas zakończenia połączenia (wartości analogiczne jak dla parametru `$initialization` opisanego wcześniej)
wartość zwracana jedna z wartości:

— TRUE – transakcja zakończona
— FALSE – wystąpił błąd podczas zakończenia transakcji

Na przesłanie danych bezpośrednio do czytnika pozwala metoda:

```
$resp=$hCard->Control($control_code , \@data);
```

Znaczenie parametrów jest szczegółowo opisane w dokumentacji danego czytnika. Metoda zwraca otrzymaną odpowiedź.

Pomocniczą metodą, pozwalającą na rozkodowanie znaczenia zwróconych bajtów statusu, jest:

```
Chipcard::PCSC::Card::ISO7816Error($sw);
```

$sw bajty statusu
wartość zwracana opis przekazanych bajtów statusu

Przykład użycia opisanych klas przedstawiono w programie poniżej (wydruk 4).

```
#!/usr/bin/perl

# dołączenie niezbędnych pakietów
use ExtUtils::testlib;
use Chipcard::PCSC;
use Chipcard::PCSC::Card;

use warnings;
use strict;

# połączenie do zarządcy zasobów
my $hContext;
# połączenie do czytnika
my $hCard;
# lista czytników
my @readers;
# status
my @status;

# pomocnicze zmienne
my $resp;
my $sw;
my $iReader;
my $rv;

# komenda GET CHALLENGE
my $GET_CHALLENGE = Chipcard::PCSC::ascii_to_array ("00 84 00 00 08");

# połączenie do zarządcy zasobów
print "Connecting PC/SC manager : ";
$hContext = new Chipcard::PCSC();
die ("failed ($Chipcard::PCSC::errno)\n")
    unless (defined $hContext);
print "success\n";

# odczytanie listy czytników w systemie
print "Readers list : ";
@readers = $hContext->ListReaders ();
die ("failed ($Chipcard::PCSC::errno)\n")
    unless (defined($readers[0]));
print "success\n";

# wydruk listy dostępnych czytników
for ($iReader=0; $iReader<=$#readers; ++$iReader)
{
    print "[".($iReader+1)."] ";
    print $readers[$iReader]."\n";
```

```
}

# wybór czytnika
do
{
    print "Select reader : ";
    $iReader = <STDIN>;
    $iReader--;
}
while ($iReader<0 || $iReader>$#readers);

# nawiązanie połączenia
print "Connecting to the reader : ";
$hCard = new Chipcard::PCSC::Card ($hContext);
die ("failed ($Chipcard::PCSC::errno)\n")
    unless (defined($hCard));
print "success\n";

# połączenie do czytnika i karty
print "Connecting to the card : ";
$rv = $hCard->Connect($readers[$iReader], $Chipcard::PCSC::SCARD_SHARE_SHARED);
unless ($rv)
{
    print "\nReconnect : ";
    $rv = $hCard->Reconnect ($Chipcard::PCSC::SCARD_SHARE_SHARED,
                             $Chipcard::PCSC::SCARD_PROTOCOL_T0 ,
                             $Chipcard::PCSC::SCARD_RESET_CARD);
    die ("failed ($Chipcard::PCSC::errno)\n")
        unless ($rv);
    print "success\n";
}
die ("Unknown protocol : $hCard->{dwProtocol}\n")
    unless ($hCard->{dwProtocol}==$Chipcard::PCSC::SCARD_PROTOCOL_T0 ||
            $hCard->{dwProtocol}==$Chipcard::PCSC::SCARD_PROTOCOL_T1 ||
            $hCard->{dwProtocol}==$Chipcard::PCSC::SCARD_PROTOCOL_RAW);
print "success\n";

# ustawienie czasu oczekiwania
print ("Setting up timeout value : ");
die ("failed ($Chipcard::PCSC::errno)\n")
    unless ($hContext->SetTimeout (50));
print "success\n";

# pobranie statusu połączenia
print "Getting status : ";
@status = $hCard->Status();
die ("failed ($Chipcard::PCSC::errno)\n")
    unless ($status[0]);
print "success\n";

print "Reader name : $status[0]\n";
print "State : $status[1]\n";
print "Current protocol : $status[2]\n";
print "ATR : " . Chipcard::PCSC::array_to_ascii ($status[3]) . "\n";

# rozpoczęcie transakcji
print ("Transaction begin : ");
die ("failed ($Chipcard::PCSC::errno)\n")
    unless ($hCard->BeginTransaction());
print "success\n";

# wysłanie komendy
print ("Exchanging data : ");
$resp = $hCard->Transmit($GET_CHALLENGE);
die ("failed ($Chipcard::PCSC::errno)\n")
    unless (defined ($resp));
print "success\n";

print "Card response : " . Chipcard::PCSC::array_to_ascii ($resp) . "\n";
```

```
    # wysłanie komendy
    $GET_CHALLENGE = "00 84 00 00 08";
    print "TransmitWithCheck : \n";
    # akceptacja dowolnej (".. ..") odpowiedzi karty zawartej w SW
    ($sw, $resp) = $hCard->TransmitWithCheck($GET_CHALLENGE, ".. ..", 1);
    warn "TransmitWithCheck: $Chipcard::PCSC::Card::Error"
        unless defined $sw;

    print "Card response : $resp (SW: $sw)\n";

    # wydruk informacji zawartej w SW
    print "ISO7816Error : ";
    print "$sw  (" . &Chipcard::PCSC::Card::ISO7816Error($sw) . ")\n";

    # zakończenie transakcji
    print ("Transaction end : ");
    die ("failed ($Chipcard::PCSC::errno)\n")
        unless ($hCard->EndTransaction($Chipcard::PCSC::SCARD_LEAVE_CARD));
    print "success\n";

    # zakończenie połączenia z kartą
    print "Disconnecting the card : ";
    $rv = $hCard->Disconnect($Chipcard::PCSC::SCARD_LEAVE_CARD);
    die ("failed ($Chipcard::PCSC::errno)\n")
        unless $rv;
    print "success\n";

    # zakończenie połączenia z czytnikiem
    print "Disconnecting reader : ";
    $hCard = undef;
    print "success\n";

    # zakończenie połączenia z zarządcą zasobów
    print "Disconnect PC/SC manager : ";
    $hContext = undef;
    print "success\n";
```

Wydruk 4. Program korzystający z interfejsu PC/SC - Perl (`PCSCExample.pl`)

6.2.3. Interfejs programisty (język Python)

Pakiet *pycsc* pozwala na korzystanie z interfejsu PC/SC z poziomu języka Python. Działa w środowisku *Linux*, *Microsoft Windows* i *Mac OS X*. Można go znaleźć w Internecie pod adresem `http://homepage.mac.com/jlgiraud/pycsc/Pycsc.html`.

Metody interfejsu dostępne są przez obiekt **pycsc**. Wykaz czytników dostępnych w systemie można uzyskać używając funkcji:

```
listReader()
```

wartość zwracana lista czytników dostępnych w systemie

Oczekiwanie na zdarzenia (włożenie karty, wyjęcie karty) realizowane jest z użyciem metody:

```
getStatusChange(Timeout, ReaderStates)
```

Timeout maksymalny czas oczekiwania na zmianę

ReaderStates parametry określające zachowanie funkcji:

- **Reader** nazwa czytnika
- **CurrentState** obserwowane stany czytnika (parametr ten przyjmuje domyślną wartość `SCARD_STATE_EMPTY`)

wartość zwracana słownik zawierający następujące klucze (oprócz kluczy przekazywanych jako parametry metody):
Atr ATR karty
EventState zdarzenie jakie wystąpiło

Utworzenie instancji klasy możliwe jest przy wykorzystaniu metody:

```
pycsc(reader, mode, protocol)
```

reader nazwa czytnika (parametr nie jest obowiązkowy, domyślnie wybierany jest pierwszy czytnik z listy dostępnych urządzeń)
mode rodzaj połączenia
— `SCARD_SHARE_SHARED` – współdzielone
— `SCARD_SHARE_EXCLUSIVE` – niewspółdzielone
protocol zalecany protokół komunikacyjny
— `SCARD_PROTOCOL_T0` – protokół T=0
— `SCARD_PROTOCOL_T1` – protokół T=1
— `SCARD_PROTOCOL_RAW` – protokół dla kart pamięciowych
wartość zwracana instancja klasy

Wykorzystanie tych metod nie wymaga wykorzystania istniejącej instancji obiektu **pycsc** (są to metody klasy).

Sprawdzenie stanu instancji obiektu związanej z danym czytnikiem wykonywane jest z użyciem metody:

```
status()
```

wartość zwracana słownik zawierający następujące klucze:
ReaderName nazwa czytnika
State stan połączenia
Protocol używany protokół
ATR ATR karty

Przesłanie komendy APDU wymaga użycia metody:

```
transmit(com, sendPCI)
```

com komenda APDU
sendPCI informacje dotyczące wykorzystywanego protokołu
wartość zwracana bufor z odpowiedzią z karty

Przerwanie połączenia możliwe jest z użyciem:

```
disconnect()
```

wartość zwracana brak

Powtórne połączenie umożliwia metoda:

```
reconnect(smode, protocol, init)
```

smode rodzaj połączenia
protocol zalecany protokół komunikacyjny
init akcja jaką należy podjąć tuż po połączeniu (zobacz opis funkcji **SCardReconnect** z 6.2.1)
wartość zwracana brak

Wymienione metody operują na instancji klasy (są to metody instancji klasy).

Poniższe metody, których działanie jest analogiczne do odpowiadających im funkcji z języka C, nie są obecnie zaimplementowane.

```
listReaderGroups()
control()
cancel()
beginTransaction()
endTransaction()
cancelTransaction()
```

Wykorzystanie interfejsu PC/SC w Python można prześledzić w przykładzie zaprezentowanym na wydruku 5.

```
#!/usr/bin/python

# dołączenie pakietu pycsc
import pycsc
# dołączenie pakietu umożliwiającego konwersję ciągów znaków na ich
# odpowiedniki w postaci heksadecymalnej i odwrotnie
import binascii

# wydruk listy czytników zainstalowanych w systemie
print "Readers list :"
for i in range(len(pycsc.listReader())):
    print "["+str(i+1)+"] "+pycsc.listReader()[i]

# wybór czytnika
reader=raw_input("Select reader :")

# nazwa wybranego czytnika
readerName = pycsc.listReader()[int(reader)-1]

# jeśli czytnik jest pusty funkcja blokuje się do czasu włożenia karty
# lub upłynięcia określonego czasu
# czas oczekiwania na kartę wynosi 5000 ms
print "Insert card..."
newState = pycsc.getStatusChange(Timeout=5000,
    ReaderStates=[{'Reader': readerName, 'CurrentState':pycsc.SCARD_STATE_EMPTY}])

# utworzenie obiektu reprezentującego kartę
card = pycsc.pycsc(readerName)

# wydruk nazwy czytnika
print "Connected to reader :" + card.status()["ReaderName"]
# stan karty
print "Card State :" + str(card.status()["State"])
# używany protokół komunikacyjny
print "Card Protocol :" + str(card.status()["Protocol"])
# wydruk ATR karty
print "Card ATR :" + binascii.b2a_hex(card.status()["ATR"])

# przesłanie do karty komendy APDU
print "APDU : GET CHALLENGE"
rapdu = card.transmit("\x00\x84\x00\x00\x08", pycsc.SCARD_PROTOCOL_T0)

# wydruk odpowiedzi otrzymanej od karty
print "Card response :" + binascii.b2a_hex(rapdu)
```

Wydruk 5. Przykład wykorzystania interfejsu PC/SC - Python (`PCSCExample.py`)

6.2.4. Interfejs programisty (język Java)

Pakiet *JPCSC* umożliwia korzystanie z interfejsu PC/SC z poziomu języka Java przy użyciu JNI (ang. *Java Native Interface*). Biblioteki natywne zostały przygotowane dla systemów

operacyjnych *Linux* oraz *Microsoft Windows*. Pakiet jest dostępny w Internecie pod adresem `http://www.linuxnet.com/middle.html`.

W skład biblioteki **com.linuxnet.jpcsc.*** wchodzi kilka klas reprezentujących logikę interfejsu PC/SC w ujęciu obiektowym:

— **Context** – klasa zawiera metody pozwalające na nawiązanie komunikacji z zarządcą zasobów PC/SC, odczytanie listy czytników oraz nawiązanie komunikacji z kartą,
— **Card** – metody tej klasy odpowiadają za nawiązanie, zerwanie połączenia z kartą oraz transmisję danych,
— **State** – obiekty tej klasy przeznaczone są do przechowywania informacji o stanach danych czytników,
— **Apdu** i **Apdu.Format** – klasy pozwalają na przechowanie i łatwą edycję komend APDU,
— **PCSC** – klasa ta zawiera podstawowe definicje stałych związanych z biblioteką PC/SC; prywatne metody są interfejsem do natywnego jądra pakietu,
— **PCSCException** – wyjątek zgłaszany przez klasy z biblioteki; korzystając z odpowiednich metod można otrzymać szczegółowy opis błędu.

Przykład wykorzystania pakietu *JPCSC* zaprezentowano na wydruku 6. Aplikacja pozwala na wybór jednego z czytników dostępnych w systemie. Następnie oczekuje na włożenie karty i przesyła przykładową komendę do niej.

Podczas uruchamiania aplikacji należy zdefiniować zmienną **java.library.path**. Dodatkowo w systemie *Linux* konieczne jest odpowiednie określenie `LD_LIBRARY_PATH`.

```
// import niezbędnych bibliotek
import com.linuxnet.jpcsc.*;
import java.io.*;

public class PCSCExample
{
    public static void main(String[] args)
    {
        // podłączenie do zarządcy zasobów PC/SC
        System.out.println("Connecting␣PC/SC␣Manager...");
        Context ctx = new Context();
        ctx.EstablishContext(PCSC.SCOPE_SYSTEM, null, null);

        // pobranie listy czytników dostępnych w systemie
        System.out.println("Readers␣list␣:");
        String[] readers = ctx.ListReaders();

        // wydruk listy czytników
        if (readers.length == 0)
        {
            System.out.println("not␣found");
            System.exit(0);
        }
        for (int i = 0; i < readers.length; i++)
        {
            System.out.println("["+Integer.toString(i+1)+"]␣" + readers[i]);
        }

        // wybór czynnika
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int reader = 0;
        do
        {
            try
            {
                System.out.print("Select␣reader␣:␣");
                String readerStr = in.readLine();
                reader = Integer.parseInt(readerStr);
            }
            catch (Exception e)
```

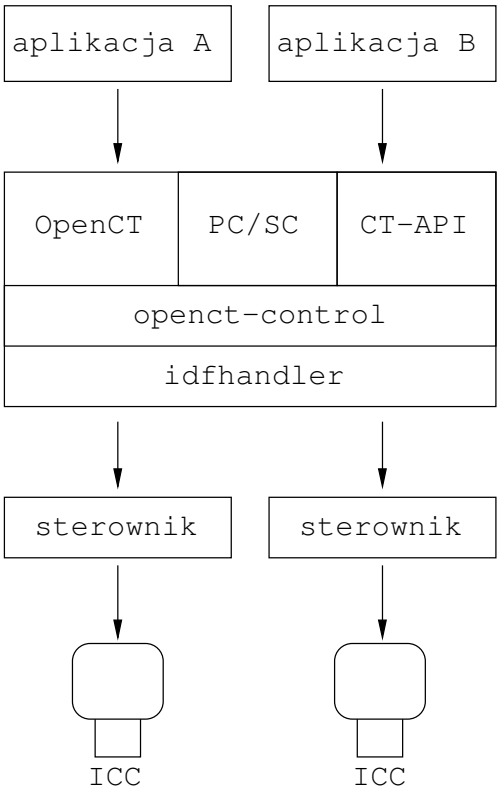
```
            {
            }
        }
        while (reader<1 || reader>readers.length);
        reader--;

        // oczekiwanie na włożenie karty do czytnika
        System.out.println("Waiting for card in " + readers[reader] + "...");
        State[] states = new State[1];
        states[0] = new State(readers[reader]);
        // oczekiwanie na zmianę statusu
        do
        {
            ctx.GetStatusChange(1000, states);
        }
        while ((states[0].dwEventState & PCSC.STATE_PRESENT) != PCSC.STATE_PRESENT);

        System.out.println("Reader state :");
        System.out.println(states[0]);

        // połączenie do karty
        System.out.println("Connecting to the card...");
        Card card = ctx.Connect(readers[reader], PCSC.SHARE_EXCLUSIVE,
                                PCSC.PROTOCOL_T1 | PCSC.PROTOCOL_T0);

        // status karty
        System.out.print("Card status :");
        State cardSatus = card.Status();
        System.out.println(cardSatus.toString());

        // przesłanie komendy APDU
        Apdu apdu = new Apdu(256);
        try
        {
            // komenda "get challenge"(CLA INS P1 P2 LE)
            final byte[] GET_CHALLENGE={(byte) 0x00, (byte) 0x84,
                                        (byte) 0x00, (byte) 0x00,
                                        (byte) 0x08};
            byte[] resp;

            apdu.set(GET_CHALLENGE);
            System.out.println("APDU :" + apdu);
            resp = card.Transmit(apdu);
            // wydruk odpowiedzi
            System.out.println("RAPDU :" + Apdu.ba2s(resp, 0, resp.length));
        }
        catch(PCSCException e)
        {
            // wydruk informacji o zaistniałym błędzie
            System.out.println("Exception! ("+e.getReason()+", "+ e.toString());
        }

        // zakończenie połączenia z kartą
        System.out.println("Disconnecting the card...");
        card.Disconnect(PCSC.LEAVE_CARD);

        // zakończenie połączenia z zarządcą zasobów PC/SC
        System.out.println("Disconnecting PC/SC manager...");
        ctx.ReleaseContext();
    }
}
```

Wydruk 6. Program wykorzystujący interfejs PC/SC - Java (`PCSCExample.java`)

6.3. OpenCT

OpenCT jest kolejnym przykładem interfejsu pozwalającego na korzystanie z czytników

kart. Celem stworzenia tej biblioteki było uproszczenie procesu tworzenia sterowników dla terminali w systemach unixowych. Z urządzeniami, które pozostają pod kontrolą *OpenCT*, można komunikować się wykorzystując natywny interfejs biblioteki lub przedstawione wcześniej CT-API oraz PC/SC (rysunek 23). Źródła można znaleźć na stronie `http://www.opensc.org`.



Rysunek 23. Komunikacja w architekturze OpenCT

Po zainstalowaniu pakietu w katalogu `/var/run/openct` przechowywane są informacje o aktualnym stanie zainstalowanych w systemie czytników. Operując prawami dostępu do tego katalogu można ograniczyć listę użytkowników, którzy mogą korzystać z tych urządzeń.

W pliku konfiguracyjnym (wydruk 7) umieszcza się informacje dotyczące ogólnej konfiguracji biblioteki, czytników na stałe zainstalowanych w systemie oraz typu *hotplug* np. poprzez interfejs USB.

```
# poziom logowania
debug=0;
# interfejsy hotplug
hotplug=yes;
# umiejscowienie aplikacji ifdhandler
ifdhandler=/usr/local/sbin/ifdhandler;
# czytniki statyczne
reader towitoko
{
    driver=towitoko;
    device=serial:/dev/ttyS0;
};
# czytniki typu hotplug (interfejs USB)
driver  egate
{
        ids=
        {
                usb:0973/0001,
```

```
        };
};
# ...
```

Wydruk 7. Plik konfiguracyjny dla środowiska OpenCT (`openct.conf`)

Aktualnie biblioteka obsługuje następujące urządzenia: Towitoko CHIPDRIVE micro, KOBIL KAAN Professional, Schlumberger e-gate, Aladdin eToken PRO oraz Eutron CryptoIdentity IT-SEC.

Aplikacjami jakie wchodzą w skład pakietu *OpenCT* są:

— **openct-control** – zarządca zasobów *OpenCT*.
Aplikacja ta pozwala na uruchomienie, zatrzymanie i sterowanie interfejsem *OpenCT*. Umożliwia również logowanie zdarzeń oraz sprawdzenie aktualnego stanu zainstalowanych czytników.

— **ifdhandler** – aplikacja będąca sterownikiem terminali.
Aplikacja wykorzystywana przez **openct-control** w celu sterowania poszczególnymi czytnikami w systemie. Obsługuje również urządzenia typu *hotplug*.

— **openct-tool** – pomocniczy program pozwalający na sprawdzenie i wykonanie prostych operacji z użyciem *OpenCT*.
Aplikacja umożliwia pobranie listy czytników zainstalowanych aktualnie w systemie, odczytanie ATR kart włożonych do określonych czytników, oczekiwanie na zainstalowanie czytnika oraz umieszczenie w nim karty. Dla kart synchronicznych możliwe jest wykonanie zrzutu zawartości pamięci, dla procesorowych - wybranie głównego katalogu w systemie plików.

Aby skorzystać z zasobów *OpenCT* poprzez interfejs PC/SC (zobacz 6.2) konieczne jest umieszczenie w pliku konfiguracyjnym **pcsc-lite** umieszczenie następującego wpisu:

```
FRIENDLYNAME      "OpenCT"
DEVICENAME        OPENCT_DEV
LIBPATH           /usr/lib/openct-ifd.so
CHANNELID         1
```

W przypadku używania CT-API (zobacz 6.1) należy jedynie dołączyć do tworzonej aplikacji odpowiednią bibliotekę.

Podstawowy interfejs programisty *OpenCT* jest przedstawiony w kolejnej sekcji.

6.3.1. Interfejs programisty

Podstawową strukturą reprezentującą informacje o urządzeniu zainstalowanym w systemie jest:

```
typedef struct ct_info
{
    char          ct_name[64];
    unsigned int  ct_slots;
    unsigned int  ct_card[OPENCT_MAX_SLOTS];
    unsigned      ct_display : 1,
                  ct_keypad  : 1;
    pid_t         ct_pid;
} ct_info_t;
```

Kolejne pola struktury reprezentują nazwę urządzenia, ilość slotów, informację o karcie w każdym ze slotów, informację o tym, czy urządzenie wyposażone jest w wyświetlacz i klawiaturę, a także numer procesu **ifdhandler** w systemie, który obsługuje dane urządzenie.

Do pobierania i operowania na danych dotyczących właściwości zainstalowanych urządzeń przeznaczonych jest kilka funkcji. Pierwsza z nich pozwala na pobranie pełnej informacji o czytnikach w systemie.

```
int ct_status(const ct_info_t **result);
```

result tablica struktur reprezentujących właściwości urządzeń
wartość zwracana ilość elementów w tablicy

Kolejne funkcje pozwalają na odczytanie informacji o urządzeniu, do którego aktualnie jesteśmy podłączeni lub znamy jego numer.

```
int ct_reader_status(ct_handle *h,
                     ct_info_t *info);
int ct_reader_info(unsigned int reader,
                   ct_info_t    *info);
```

h kontekst połączenia
reader numer czytnika
info struktura zawierająca informacje o urządzeniu
wartość zwracana pierwsza z funkcji zawsze zwraca `IFD_SUCCESS`; druga zwraca wartość zdefiniowaną jako `IFD_ERROR_GENERIC` w przypadku błędu albo `IFD_SUCCESS`

Nawiązanie połączenia z urządzeniem (uzyskanie kontekstu) oraz późniejsze zamknięcie komunikacji z czytnikiem możliwe jest przy wykorzystaniu funkcji:

```
ct_handle *ct_reader_connect(unsigned int reader);
void ct_reader_disconnect(ct_handle *h);
```

reader numer czytnika w systemie
h kontekst połączenia
wartość zwracana pierwsza funkcja zwraca kontekst połączenia

Sprawdzenie aktualnego stanu slotu w czytniku (czy włożona jest tam karta), zerowanie karty lub oczekiwanie na jej włożenie do czytnika umożliwiają funkcje:

```
int ct_card_status(ct_handle    *h,
                   unsigned int slot,
                   int          *status);
int ct_card_reset(ct_handle     *h,
                  unsigned int  slot,
                  void          *atr,
                  size_t        atr_len);
int ct_card_request(ct_handle    *h,
                    unsigned int slot,
                    unsigned int timeout,
                    const char   *message,
                    void         *atr,
                    size_t       atr_len);
```

h kontekst połączenia
slot numer slotu w danym czytniku
status stan karty w czytniku; wartość może być kombinacją następujących flag:
— `IFD_CARD_PRESENT` – karta obecna w czytniku
— `IFD_CARD_STATUS_CHANGED` – zmienił się stan karty
atr bufor zawierający ATR

atr_len długość bufora z ATR
timeout maksymalny czas oczekiwania
message tekst, który ma zostać wyświetlony na wyświetlaczu terminala
wartość zwracana ujemne wartości oznaczają błąd, wartość dodatnia to długość ATR; dla pierwszej z funkcji wartość `IFD_SUCCESS` oznacza prawidłowe zakończenie

Sterowanie dostępnością slotu dla innych aplikacji w danym czytniku jest możliwe przy wykorzystaniu funkcji:

```
int ct_card_lock(ct_handle      *h,
                 unsigned int   slot,
                 int            type,
                 ct_lock_handle *lock);
int ct_card_unlock(ct_handle      *h,
                   unsigned int   slot,
                   ct_lock_handle lock);
```

h kontekst połączenia
slot numer slotu w danym czytniku
type rodzaj blokady
— `IFD_LOCK_SHARED` – karty w danym slocie mogą być używane wyłącznie przez aplikację korzystającą z danego slotu
— `IFD_LOCK_EXCLUSIVE` – inne aplikacje mają dostęp do danego slotu
lock zmienna reprezentująca blokadę
wartość zwracana wartość ujemna w przypadku błędu, wartość `IFD_SUCCESS` po poprawnym wykonaniu

Przesłanie komendy do karty i uzyskanie odpowiedzi realizuje funkcja:

```
int ct_card_transact(ct_handle    *h,
                     unsigned int slot,
                     const void   *apdu,
                     size_t       apdu_len,
                     void         *recv_buf,
                     size_t       recv_len);
```

h kontekst połączenia
slot numer slotu w danym czytniku
apdu, apdu_len bufor z komendą APDU i jej długość
recv_buf, recv_len bufor odbiorczy i jego długość
wartość zwracana wartość ujemna w przypadku błędu, wartość dodatnia (ilość danych odpowiedzi z karty) po poprawnym wykonaniu

Na weryfikację kodu PIN pozwala funkcja:

```
int ct_card_verify(ct_handle    *h,
                   unsigned int slot,
                   unsigned int timeout,
                   const char   *prompt,
                   unsigned int pin_encoding,
                   unsigned int pin_length,
                   unsigned int pin_offset,
                   const void   *send_buf,
                   size_t       send_len,
                   void         *recv_buf,
                   size_t       recv_len);
```

h kontekst połączenia
slot numer slotu w danym czytniku
timeout maksymalny czas oczekiwania
prompt napis jaki ma zostać wyświetlony przy zapytaniu o PIN
pin_encoding sposób zapisu kodu PIN
— `IFD_PIN_ENCODING_BCD` – kodowanie BCD
— `IFD_PIN_ENCODING_ASCII` – kodowanie ASCII
pin_length długość kodu PIN
pin_offset offset w buforze dla kodu PIN
send_buf, send_len bufor z danymi do przesłania i jego wielkość
recv_buf, recv_len bufor na dane (odpowiedź karty) i jego maksymalna wielkość
wartość zwracana wartość ujemna w przypadku błędu, wartość dodatnia (ilość danych odpowiedzi z karty) po poprawnym wykonaniu

Dla kart synchronicznych istnieją funkcje umożliwiające odczyt i zapis ich pamięci:

```
int ct_card_read_memory(ct_handle       *h,
                        unsigned int    slot,
                        unsigned short  address,
                        void            *recv_buf,
                        size_t          recv_len);
int ct_card_write_memory(ct_handle      *h,
                         unsigned int   slot,
                         unsigned short address,
                         const void     *send_buf,
                         size_t         send_len);
```

h kontekst połączenia
slot numer slotu w danym czytniku
address początkowy adres pamięci do odczytu (zapisu)
recv_buf, recv_len bufor na dane i jego maksymalna wielkość
send_buf, send_len bufor z danymi do przesłania i jego wielkość
wartość zwracana wartość ujemna w przypadku błędu, wartość `IFD_SUCCESS` albo dodatnia po poprawnym wykonaniu

Na wydruku 8 zamieszczono przykład wykorzystania interfejsu *OpenCT*. Aplikacja drukuje na ekranie dostępne czytniki, prosi o wybranie jednego z nich, a następnie pobiera ATR i odpowiedź na komendę GET CHALLENGE dla karty umieszczonej w pierwszym slocie.

```
// dołączenie niezbędnych plików nagłówkowych
#include <stdio.h>
#include <stdlib.h>

// prototypy funkcji OpenCT
#include <openct.h>

int main(int argc, char **argv)
{
    // komenda GET CHALLENGE
    unsigned char GET_CHALLENGE[] = {0x00, 0x84, 0x00, 0x00, 0x08};

    // bufory na ATR i odpowiedź od karty
    unsigned char atr[33];
    unsigned char res[10];

    // kontekst połączenia
    ct_handle       *h;
    // blokada
```

```
20      ct_lock_handle  lock;

        // pomocnicze zmienne
        int reader, i, rsp;

25      // wydruk czytników dostępnych w systemie
        for (i=0; i<OPENCT_MAX_READERS; i++)
        {
            ct_info_t info;
            // pobranie informacji o czytniku
30          if (ct_reader_info(i, &info) < 0)
                continue;
            printf("Reader␣%02d:␣", i);
            // wydruk nazwy czytnika
            printf("%s\n", info.ct_name);
35      }

        // wybór czytnika do dalszych operacji
        do
        {
40          printf("Select␣reader␣:␣");
            scanf("%d", &reader);
        }
        while (reader>OPENCT_MAX_READERS || reader<=0);

45      // podłączenie do czytnika
        if (!(h = ct_reader_connect(reader)))
        {
            printf("Failed␣to␣connect␣%02d\n", reader);
            return 1;
50      }

        // nałożenie blokady
        if ((rsp=ct_card_lock(h, 0, IFD_LOCK_EXCLUSIVE, &lock))<0)
        {
55          printf("Fatled␣to␣lock␣the␣card␣[%d]\n", rsp);
            return 1;
        }

        // zerowanie karty
60      rsp=do_reset(h, atr, sizeof(atr));

        // wydruk ATR
        printf("ATR␣[%d]␣=␣", rsp);
        for (i=0; i<rsp; ++i)
65      {
            printf("%02X␣", atr[i]);
        }
        printf("\n");

70      // przesłanie komedy do karty
        rsp=ct_card_transact(h, 0, GET_CHALLENGE, sizeof(GET_CHALLENGE),
                             res, sizeof(res));
        if (rsp<0)
        {
75          printf("Fatled␣to␣send␣GET_CHALLENGE␣[%d]\n", rsp);
            return 1;
        }

        // wydruk odpowiedzi
80      printf("RAPDU␣[%d]␣=␣", rsp);
        for (i=0; i<rsp; ++i)
        {
            printf("%02X␣", res[i]);
        }
85      printf("\n");

        // zdjęcie blokady
        ct_card_unlock(h, 0, lock);
```

```
        if (rsp<0)
90      {
            printf("Failed to unlock [%d]\n", rsp);
            return 1;
        }

95      // zamknięcie połączenia
        ct_reader_disconnect(h);

        return 0;
    }
```

Wydruk 8. Aplikacja korzystająca z interfejsu OpenCT (`OpenCTExample.c`)

6.3.2. Projektowanie sterowników

Tworzenie sterowników dla nowych urządzeń z użyciem *OpenCT* jest uproszczone, gdyż pozwala skupić się wyłącznie na problemach związanych wyłącznie z komunikacją na poziomie fizycznym. Obsługa najważniejszych właściwości (takich jak operacje związane z protokołami komunikacyjnymi T=0 lub T=1) jest już zaimplementowana. Dzięki temu implementacja sprowadza się do uzupełnienia funkcji służących do aktywacji i dezaktywacji urządzenia, wysyłania i odbierania danych, sprawdzania stanu oraz zerowania karty. Stworzonymi sterownikami zarządza aplikacja **ifdhandler**.

Więcej informacji można uzyskać studiując pliki w katalogu `/src/idf` dystrybucji źródeł *OpenCT*. Znajdują się tam również przykładowe implementacje sterowników (zarówno USB jak i podłączanych poprzez port szeregowy).

6.4. OpenCard Framework

OpenCard Consortium, organizacja rozwijająca *OpenCard Framework*, skupia kilkanaście firm związanych z przemysłem kartowym. Biblioteka stworzona została w języku Java. Oznacza to, że charakteryzuje się obiektowym podejściem do problematyki obsługi kart w czytnikach. Oprogramowanie oraz szczegółowa dokumentacja znajdują się w sieci Internet pod adresem `http://www.opencard.org/`. Instalacja (wymagane jest poprawnie skonfigurowane środowisko Java) sprowadza się do uruchomienia programu, który wypakuje biblioteki do wybranego katalogu.

Podstawowym założeniem twórców tego rozwiązania było uniezależnienie się od:

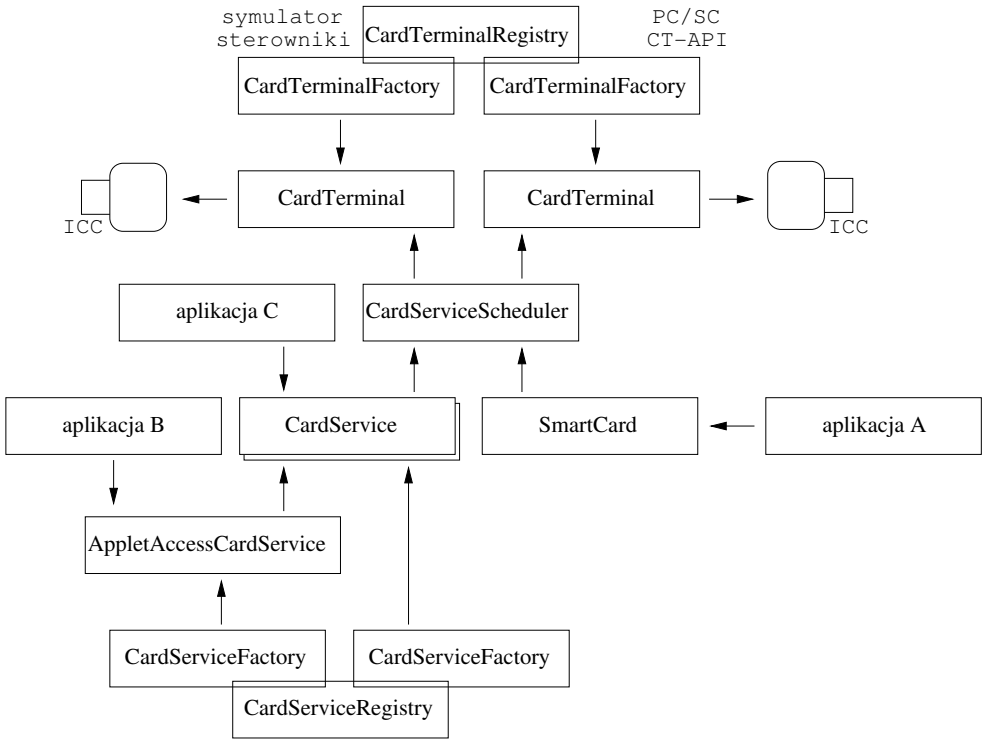
— producentów czytników kart – wyróżnia się klasy obsługujące warstwę czytników *CardTerminal*,
— systemów operacyjnych kart – wyróżnia się klasy obsługujące warstwę systemu operacyjnego karty *CardService*,
— dostawców aplikacji kartowych – wyróżnia się klasy obsługujące warstwę aplikacji kartowych *ApplicationManagementCardService*.

Oznacza to, że dla stworzonej aplikacji korzystającej z karty warstwy te są przezroczyste tj. może ona współpracować zarówno z rzeczywistą kartą jak i z symulatorem nie wymagając przy tym żadnych zmian w kodzie źródłowym.

Na rysunku 24 pokazano główne elementy *OpenCard Framework*.

Każdy z terminali reprezentowany jest przez instancję klasy *CardTerminal*. Odpowiednie sterowniki dostarczane sią przez obiekty typu *CardTerminalFactory*, a konfiguracja zarządzana przez *CardTerminalRegistry*. Mogą to być zarówno urządzenia dostępne przez interfejs PC/SC czy CT-API jak i symulatory wykorzystujące protokół TCP/IP do komunikacji. Obiekty *CardTerminal* mogą przekazywać zdarzenia do wszystkich nasłuchujących komponentów biblioteki.

Dostęp do kart inteligentnych możliwy jest na dwa główne sposoby:



Rysunek 24. Komunikacja w architekturze OCF

— poprzez obiekt *SmartCard* stworzony bezpośrednio przez *CardTerminal* (pozwala na przesyłanie komend APDU),
— wykorzystując obiekty *CardService*, które opisują wysokopoziomowe API dla kart np.: *AppletAccessCardService*, który pozwala na pobranie listy aplikacji umieszczonych w karcie.

Instancje klas *CardService* tworzone są przez obiekty *CardServiceFactory*. Podobnie jak dla czytników konfiguracja zarządzana jest przez rejestr, który reprezentuje klasa *CardServiceRegistry*.

Jak już wspomniano, biblioteka pracuje z czytnikami, dla których stworzono odpowiednie klasy je obsługujące. Definiuje je się w pliku konfiguracyjnym, którego przykład pokazano na wydruku 9.

OpenCard.services definiuje klasy obsługujące systemy operacyjne kart. W przykładzie wymieniono **PassThruCardServiceFactory** zgodną ze wszystkimi kartami inteligentnymi i pozwalającą na komunikację na poziomie komend APDU (niskopoziomową). *OpenCard.terminals* określa klasy obsługujące czytniki. W przykładzie wymieniono klasy obsługujące czytniki zgodne z PC/SC (zobacz 6.2) oraz czytnik GemPC 410 podłączony do portu COM1. *OpenCard.trace* pozwala zdefiniować poziom szczegółowości raportowania zdarzeń w bibliotece (szczegółowy opis wartości znajduje się w dokumentacji).

```
OpenCard.services = \
opencard.opt.util.PassThruCardServiceFactory
OpenCard.terminals = \
com.ibm.opencard.terminal.pcsc10.Pcsc10CardTerminalFactory \
com.gemplus.opencard.terminal.GemplusCardTerminalFactory|gempc410_com1|GemPC410|COM1
OpenCard.trace = opencard:5
```

Wydruk 9. Przykładowa konfiguracja (`opencard.properties`)

6.4.1. Interfejs programisty

Klasy obsługujące terminale w dużej części oparte są na interfejsie JNI (ang. *Java Native Interface*) wymagającym bibliotek natywnych dla określonej platformy. Wynika to z różnic na poziomie systemu operacyjnego (np. programowanie portu RS232).

Kolejne przykłady prezentują możliwości biblioteki *OpenCard Framework*. Na wydruku 10 zaprezentowano najprostszy, a zarazem najbardziej bezpieczny sposób programowania. Polega on na samodzielnym odczycie dostępnych zasobów systemowych oraz wyborze jednego z nich.

Klasa **CardTerminalRegistry** przechowuje informację o czytnikach zarejestrowanych w systemie (na podstawie informacji zawartych w pliku konfiguracyjnym). Korzystając z metody **getCardTerminals()** można odczytać informację o terminalach. Możliwa jest również dynamiczna zmiana zawartości rejestru tj. dodawanie lub usuwanie czytników. Rejestr jest automatycznie uaktualniany poprzez wywołanie metody **SmartCard.start()**. Metoda ta inicjalizuje bibliotekę *OpenCard Framework* i umożliwia korzystanie z niej.

CardTerminal reprezentuje czytnik wraz z jego slotami (w przypadku urządzeń, które obsługują jednocześnie kilka kart). Najczęściej jest to jeden slot (tradycyjne czytniki na jedną kartę). Poprzez klasę **SlotChannel** możliwa jest komunikacja z samą kartą. Obiekt klasy **CardID**, który tworzony jest po włożeniu karty do slotu zawiera odpowiedź na zerowanie karty. Komendy do karty można przesyłać tworząc obiekty klasy **CommandAPDU** reprezentujące instrukcję. Odpowiedź karty zawarta jest w polach klasy **ResponseAPDU**.

```
// import niezbędnych klas
import java.util.Enumeration;
import java.util.Properties;

import opencard.core.service.SmartCard;
import opencard.core.service.CardServiceException;

import opencard.core.terminal.CardTerminalRegistry;
import opencard.core.terminal.CardTerminal;
import opencard.core.terminal.SlotChannel;
import opencard.core.terminal.CardID;
import opencard.core.terminal.CommandAPDU;
import opencard.core.terminal.ResponseAPDU;
import opencard.core.terminal.CardTerminalException;

import opencard.core.util.HexString;
import opencard.core.util.OpenCardPropertyLoadingException;

public class OCFExample1
{
    public static void main (String[] args)
    {
        // rejestr czytników w systemie
        CardTerminalRegistry ctr;

        // lista terminali
        Enumeration terminals;

        // terminal (czytnik)
        CardTerminal terminal = null;

        // aktywny slot terminala (czytnika)
        SlotChannel sch;

        // komenda "get challenge"(CLA INS P1 P2 LE)
        final byte[] GET_CHALLENGE={(byte) 0x00, (byte) 0x84,
                                    (byte) 0x00, (byte) 0x00,
                                    (byte) 0x08};

        try
        {
```

```
        // próba uruchomienia usług związanych z kartami inteligentnymi
        SmartCard.start();

45      // pobranie rejestru terminali
        ctr = CardTerminalRegistry.getRegistry();

        // pobranie listy terminali z rejestru
        terminals = ctr.getCardTerminals();
50
        // dla wszystkich terminali z rejestru
        while (terminals.hasMoreElements())
        {
            terminal = (CardTerminal) terminals.nextElement();
55
            // dane o terminalu
            System.out.println("Address : "+terminal.getAddress());
            System.out.println("Name    : "+terminal.getName());
            System.out.println("Type    : "+terminal.getType());
60
            // pobranie ilości slotów w terminalu
            int slots = terminal.getSlots();

            System.out.println("Slots number : "+slots);
65
            // wydruk właściwości terminala
            Properties termProps = terminal.features ();
            if (termProps!=null)
            {
70              System.out.println("Terminal properties : ");
                termProps.list(System.out);
            }
            else
                System.out.println("No terminal properties.");
75
            // sprawdzenie każdego slotu terminala
            for (int slotID=0; slotID<slots; slotID++)
            {
                // sprawdzenie dostępności slotu
80              System.out.println("Slot " + slotID +
                        " channel available : " +
                        (terminal.isSlotChannelAvailable(slotID) ?
                        "yes" : "no"));
                try
85              {
                    // otwarcie slotu
                    sch=terminal.openSlotChannel(slotID);

                    // sprawdzenie czy w slocie znajduje się karta
90                  System.out.println("Card present : " +
                            (terminal.isCardPresent(slotID) ?
                             "yes" : "no") );

                    // informacje o karcie
95                  CardID cid = sch.getCardID();

                    // wydruk ATR karty
                    System.out.println ("ATR : " +
                            HexString.hexify(cid.getATR()));
100
                    // utworzenie komendy GET CHALLENGE
                    CommandAPDU apdu = new CommandAPDU(GET_CHALLENGE);
                    // przesłanie komendy do karty
                    ResponseAPDU rpdu = sch.sendAPDU(apdu);
105
                    // wydruk otrzymanych wyników
                    System.out.println("Response data : " +
                            HexString.hexify(rpdu.data()));

110                 System.out.println("Response SW : " +
```

```
                                    HexString.hexifyShort(rpdu.sw()));
                        }
                        catch(CardTerminalException ex)
                        {
115                         System.out.println("CardTerminalException : ");
                            System.out.println(ex.getMessage());
                        }
                        System.out.println();
                    }
120             }
                // zamknięcie usług związanych z kartami inteligentnymi
                SmartCard.shutdown();
            }
            // wyjątek związany z usługami systemowymi
125         catch (CardServiceException ex)
            {
                System.out.println("CardServiceException : ");
                System.out.println(ex.getMessage());
            }
130         // wyjątek zgłaszany przy błędach terminala
            catch (CardTerminalException ex)
            {
                System.out.println("CardTerminalException : ");
                System.out.println(ex.getMessage());
135         }
            // wyjątek zgłaszany w sytuacji nieprawidłowego
            // pliku konfiguracyjnego lub jego braku
            catch (OpenCardPropertyLoadingException ex)
            {
140             System.out.println("OpenCardPropertyLoadingException : ");
                System.out.println(ex.getMessage());
            }
            // w przypadku braku dostępu do odpowiednich klas
            catch (ClassNotFoundException ex)
145         {
                System.out.println("ClassNotFoundException : ");
                System.out.println(ex.getMessage());
            }
        }
150 }
```

Wydruk 10. Wykorzystanie biblioteki OCF (`OCFExample1.java`)

Biblioteka jest zaprojektowana w taki sposób, aby możliwe było generowanie i przechwytywanie zdarzeń zaistniałych w systemie np. włożenie karty lub jej usunięcie ze slotu terminala. Przykład z wydruku 11 wykorzystuje klasę **CardRequest**, która umożliwia oczekiwanie na określone zdarzenie (w tym wypadku na włożenie karty).

Taki sposób programowania może jednak stwarzać problemy. Przykładowo w systemie *Microsoft Windows* po niepoprawnym odinstalowaniu czytnika (tj. nie ma go jako fizycznego urządzenia, ale pozostał wpis w rejestrach systemowych) program może zachowywać się w sposób nieokreślony. Dotyczy to czytników obsługiwanych poprzez PC/SC.

```
// import niezbędnych klas
import opencard.core.service.SmartCard;
import opencard.core.service.CardRequest;
import opencard.core.service.CardServiceException;
5
import opencard.core.terminal.CardID;
import opencard.core.terminal.CommandAPDU;
import opencard.core.terminal.ResponseAPDU;
import opencard.core.terminal.CardTerminalException;
10
import opencard.opt.util.PassThruCardService;

import opencard.core.util.HexString;
import opencard.core.util.OpenCardPropertyLoadingException;
```

```
15
public class OCFExample2
{
    public static void main (String[] args)
    {
20      // komenda APDU
        CommandAPDU apdu;

        // odpowiedź APDU
        ResponseAPDU rpdu;
25
        // komenda "get challenge"(CLA INS P1 P2 LE)
        final byte[] GET_CHALLENGE={(byte) 0x00, (byte) 0x84,
                                    (byte) 0x00, (byte) 0x00,
                                    (byte) 0x08};
30
        try
        {
            // uruchomienie usług związanych z kartami inteligentnymi
            SmartCard.start();
35
            // klasa reprezentuje kartę na jaką czeka aplikacja
            CardRequest cr = new CardRequest(CardRequest.ANYCARD,
                    null, PassThruCardService.class);

40          // ustawienie czasu oczekiwania
            cr.setTimeout(0);

            // oczekiwanie na włożenie karty (w sekundach)
            SmartCard sc = SmartCard.waitForCard(cr);
45
            if (sc != null)
            {
                // klasa obsługująca sesję z kartą
                PassThruCardService ptcs = (PassThruCardService)
50                      sc.getCardService(PassThruCardService.class, true);

                // pobranie informacji o karcie
                CardID cid = sc.getCardID ();

55              // wydruk ATR karty
                System.out.println ("ATR : " +
                        HexString.hexify(cid.getATR()));

                // przesłanie komendy do karty
60              apdu = new CommandAPDU(GET_CHALLENGE);
                rpdu = ptcs.sendCommandAPDU(apdu);

                // wydruk otrzymanych wyników
                System.out.println("Response data : " +
65                      HexString.hexify(rpdu.data()));

                System.out.println("Response SW : " +
                        HexString.hexifyShort(rpdu.sw()));
            }
70          else
                System.out.println("Card not inserted!");
            // zamknięcie usług związanych z kartami inteligentnymi
            SmartCard.shutdown();
        }
75      // wyjątek związany z usługami systemowymi
        catch (CardServiceException ex)
        {
            System.out.println("CardServiceException : ");
            System.out.println(ex.getMessage());
80      }
        // wyjątek zgłaszany przy błędach terminala
        catch (CardTerminalException ex)
        {
```

```
            System.out.println("CardTerminalException : ");
            System.out.println(ex.getMessage());
        }
        // wyjątek zgłaszany w sytuacji nieprawidłowego
        // pliku konfiguracyjnego lub jego braku
        catch (OpenCardPropertyLoadingException ex)
        {
            System.out.println("OpenCardPropertyLoadingException : ");
            System.out.println(ex.getMessage());
        }
        // w przypadku braku dostępu do odpowiednich klas
        catch (ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException : ");
            System.out.println(ex.getMessage());
        }
    }
}
```

Wydruk 11. Wykorzystanie biblioteki OCF (`OCFExample2.java`)

Ostatni przykład (wydruk 12) ukazuje sposób programowania zdarzeniowego. Jest to zarazem w pełni obiektowe wykorzystanie biblioteki *OpenCard Framework*. Niestety podobnie jak w poprzednim przypadku ta metoda programowania przy niepoprawnej konfiguracji środowiska kart inteligentnych w systemie może doprowadzić do wadliwego działania aplikacji.

Klasa **CardTerminal**, która zarządza obiektami SlotChannel, reprezentującymi jej sloty, generuje zdarzenia podczas włożenia lub usunięcia karty. Przekazywane są one do odpowiednich klas, które były zarejestrowane z użyciem metod klasy **EventGenerator**.

W celu przechwytywania zdarzeń należy zaimplementować metody interfejsu **CTListener**.

```
// import niezbędnych klas
import opencard.core.service.SmartCard;
import opencard.core.service.CardServiceException;

import opencard.core.terminal.CardTerminal;
import opencard.core.terminal.SlotChannel;
import opencard.core.terminal.CardID;
import opencard.core.terminal.CardTerminalException;
import opencard.core.terminal.InvalidSlotChannelException;

import opencard.core.event.CTListener;
import opencard.core.event.CardTerminalEvent;
import opencard.core.event.EventGenerator;

import opencard.core.util.HexString;
import opencard.core.util.OpenCardPropertyLoadingException;

public class OCFExample3
{
    // obiekt synchronizujący zdarzenia
    private static Object SCMonitor = "Smart Card Event";

    // implementacja interfejsu CTListener
    static class CardListener implements CTListener
    {
        public void cardInserted (CardTerminalEvent ctEvent)
        {
            System.out.println ("Card inserted.");

            try
            {
                // pobranie instancji terminala
                CardTerminal terminal = ctEvent.getCardTerminal();

                // pobranie numeru slotu
                int slotID = ctEvent.getSlotID();
```

```
                    // otwarcie slotu
                    SlotChannel sch=terminal.openSlotChannel(slotID);

                    // informacje o karcie
                    CardID cid = sch.getCardID();

                    // wydruk ATR karty
                    System.out.println ("ATR : " +
                            HexString.hexify(cid.getATR()));
                }
                // wyjątek zgłaszany przy błędach slotu
                catch (InvalidSlotChannelException ex)
                {
                    System.out.println("InvalidSlotChannelException : ");
                    System.out.println(ex.getMessage());
                }
                // wyjątek zgłaszany przy błędach terminala
                catch (CardTerminalException ex)
                {
                    System.out.println("CardTerminalException : ");
                    System.out.println(ex.getMessage());
                }

                // zgłoszenie zdarzenia
                synchronized (SCMonitor)
                {
                    SCMonitor.notifyAll();
                }
            }

            public void cardRemoved (CardTerminalEvent ctEvent)
            {
                System.out.println("Card removed.");

                // zgłoszenie zdarzenia
                synchronized (SCMonitor)
                {
                    SCMonitor.notifyAll();
                }
            }
        }

        public static void main (String[] args)
        {
            try
            {
                // uruchomienie usług związanych z kartami inteligentnymi
                SmartCard.start();

                // rejestracja zdarzenia
                EventGenerator.getGenerator().addCTListener(new CardListener());

                synchronized (SCMonitor)
                {
                    System.out.println("Waiting for event...");
                    try
                    {
                        // oczekiwanie na zdarzenie włożenia lub wyjęcia karty
                        SCMonitor.wait();
                    }
                    catch (InterruptedException ex)
                    {
                        System.out.println("InterruptedException : ");
                        System.out.println(ex.getMessage());
                    }
                }

                SmartCard.shutdown();
```

```
            }
            // wyjątek związany z usługami systemowymi
            catch (CardServiceException ex)
            {
                System.out.println("CardServiceException : ");
                System.out.println(ex.getMessage());
            }
            // wyjątek zgłaszany przy błędach terminala
            catch (CardTerminalException ex)
            {
                System.out.println("CardTerminalException : ");
                System.out.println(ex.getMessage());
            }
            // wyjątek zgłaszany w sytuacji nieprawidłowego
            // pliku konfiguracyjnego lub jego braku
            catch (OpenCardPropertyLoadingException ex)
            {
                System.out.println("OpenCardPropertyLoadingException : ");
                System.out.println(ex.getMessage());
            }
            // w przypadku braku dostępu do odpowiednich klas
            catch (ClassNotFoundException ex)
            {
                System.out.println("ClassNotFoundException : ");
                System.out.println(ex.getMessage());
            }
        }
    }
```

Wydruk 12. Wykorzystanie biblioteki OCF (`OCFExample3.java`)

Jak już wcześniej wspomniano oprócz niskopoziomowych odwołań do kart (poprzez komendy APDU) jakie zaprezentowano w przykładach możliwe jest również wykorzystanie klas obsługujących systemy operacyjne kart. Tworzone są one zazwyczaj przez producentów kart. *OpenCard Framework* dostarcza jedynie podstawowych interfejsów dla systemów operacyjnych kart. Na przykład **FileAccessCardService** dostarcza metod umożliwiających operacje na plikach zgodne z ISO 7816-4. Poniżej zaprezentowano fragment kodu, który wykorzystuje możliwości tej klasy. Obiekt **fs** pozwala na wykorzystanie wysokopoziomowych odwołań do karty. Przykładowo może to być pobranie ścieżki do głównego pliku karty.

```
FileAccessCardService fs = (FileAccessCardService)
    card.getCardService(FileAccessCardService.class);
CardFilePath rootpath = fs.getRoot();
```

Interfejs ten pozwala również na tworzenie, usuwanie oraz modyfikację plików. Ścieżki do plików reprezentowane są przez obiekty klas **CardFilePath** a same pliki przez klasę **CardFile**.

6.4.2. Aplety dla przeglądarek

Wykorzystanie języka Java ma swoje szczególne znaczenie w Internecie. Aplety dla przeglądarek potrafią znacząco rozszerzyć ich możliwości, a przygotowane i skonfigurowane w odpowiedni sposób nie stanowią zagrożenia dla bezpieczeństwa.

Korzystając z *OpenCard Framework* można tworzyć aplety umożliwiające dostęp do czytnika klienta przeglądarki. Ponieważ konieczny będzie dostęp do zewnętrznych urządzeń z poziomu przeglądarki należy ustalić (w zależności od rodzaju przeglądarki) jego rodzaj. Dla przeglądarek zgodnych z *Netscape* możliwe jest to poprzez wykonanie kodu:

```
opencard.core.util.SystemAccess sys =
    new opencard.opt.netscape.NetscapeSystemAccess();
opencard.core.util.SystemAccess.setSystemAccess(sys);
```

natomiast dla dla *Internet Explorer*:

```
opencard.core.util.SystemAccess sys =
    new opencard.opt.ms.MicrosoftSystemAccess();
opencard.core.util.SystemAccess.setSystemAccess(sys);
```

W zależności od wykorzystywanej przeglądarki konieczne jest umieszczenie pliku z konfiguracją w odpowiednim miejscu. Najczęściej jest to katalog domowy użytkownika lub katalog, w którym zainstalowana została maszyna wirtualna.

Pakiet zawierający aplet powinien być podpisany cyfrowo (służą do tego standardowe narzędzia dostarczane z maszyną wirtualną). Użytkownik, poprzez edycję pliku `java.policy`, powinien umożliwić pakietom podpisanym przez określoną osobę dostęp do plików z konfiguracją *OpenCard Framework*, zmiennych systemowych oraz możliwość ładowania bibliotek natywnych.

```
grant signedBy "Osoba"
{
    // odczyt zmiennych systemowych
    permission java.util.PropertyPermission "*","read,write";
    // odczyt pliku opencard.properties
    permission java.io.FilePermission
        "${java.home}/lib/opencard.properties", "read";
    permission java.io.FilePermission
        "${user.home}/.opencard.properties", "read";
    permission java.io.FilePermission
        "${user.dir}/opencard.properties", "read";
    permission java.io.FilePermission
        "${user.dir}/.opencard.properties", "read";
    // inne uprawnienia
    permission java.lang.RuntimePermission "loadLibrary.*";
    permission java.lang.RuntimePermission "reflect.declared.*";
};
```

Zastosowaniem apletów internetowych może być np. możliwość wykorzystania tokenu dostępowego w aplikacjach internetowych, zdalne doładowanie elektronicznej portmonetki itp.

6.5. Security and Trust Services (SATSA)

Security and Trust Services to opcjonalny pakiet przeznaczony dla *Java 2 Micro Edition* umożliwiający korzystanie z bezpiecznych urządzeń (ang. *security elements*), którymi w szczególności mogą być karty inteligentne. Biblioteka pozwala na integrację tych urządzeń lub ich programowych implementacji w celu stworzenia bezpiecznego środowiska. Rozwiązanie powstało głównie z myślą wykorzystania kart procesorowych z telefonów komórkowych.

Na stronie `http://java.sun.com/products/satsa/` dostępne jest oprogramowanie wraz z dokumentacją.

6.5.1. Interfejs programisty

SATSA zawiera cztery podstawowe zestawy pakietów:

— SATSA-APDU – definiuje interfejs przeznaczony do komunikacji na poziomie komend APDU,
— SATSA-JCRMI – definiuje interfejs przeznaczony do komunikacji z użyciem RMI,
— SATSA-PKI – interfejs do realizacji operacji podpisu elektronicznego,
— SATSA-CRYPTO – podstawowe algorytmy kryptograficzne.

SATSA-APDU (pakiet **javax.microedition.apdu**) pozwala tworzyć obiekty klasy **APDUConnection** umożliwiające komunikację na poziomie komend APDU zgodnie z normą ISO 7816-4. Na wydruku 13 zaprezentowano fragment aplikacji korzystający z tego pakietu.

```
// komenda APDU
final byte [] commandAPDU={0x00, 0x84, 0x00, 0x00, 0x08}
// bufor na odpowiedź karty
byte[] responseAPDU[10];
// obiekt reprezentujący połączenie
APDUConnection acn = null;
try
{
    // utworzenie obiektu APDUConnection (na podstawie AID apletu)
    acn = (APDUConnection)
        Connector.open("apdu:0;target=A0.0.0.0.1.1.1.1.1.1");
    // wysłanie komendy do karty i odebranie odpowiedzi
    responseAPDU = acn.exchangeAPDU(commandAPDU);
    // ...
}
catch (IOException e)
{
    // ...
}
finally
{
    // ...
    if(acn != null)
    {
        // zamknięcie połączenia
        acn.close();
    }
}
```

Wydruk 13. Security and Trust Services (`SATSAExample.java`)

Pakiet **javax.microedition.jcrmi** (SATSA-JCRMI) umożliwia komunikację poprzez RMI. Odpowiednich metod dostarcza klasa **JavaCardRMIConnection**. Użycie SATSA-JCRMI zaprezentowano na wydruku 31 (strona 105).

6.6. SCEZ

SCEZ jest prostą biblioteką pozwalającą na korzystanie z czytników i kart w systemach *Linux*, *Windows*, *FreeBSD* oraz *PalmOS*. Jej źródła oraz informacje o aktualnie wspieranych urządzeniach znajdują się pod adresem `http://www.franken.de/crypt/scez.html`.

6.6.1. Interfejs programisty

Rozpoczęcie i zakończenie korzystania z biblioteki sygnalizowane jest poprzez wywołanie odpowiedniej funkcji:

```
int scInit();
int scEnd();
```

Najważniejszymi strukturami danych występującymi w bibliotece *SCEZ* są:

— `SC_READER_INFO` – przechowuje informacje o czytniku i jego parametrach,
— `SC_CARD_INFO` – zawiera informacje dotyczące karty,
— `SC_APDU` – struktura przeznaczona na przechowywanie komendy i odpowiedzi APDU.

Alokacja pamięci oraz jej zwalnianie dla wymienionych struktur możliwe są przy użyciu odpowiednich funkcji. Przykładowo dla `SC_APDU` mają one następującą postać (analogiczna deklaracja jest dla innych obiektów):

```
SC_APDU *scGeneralNewAPDU();
void scGeneralFreeAPDU(SC_APDU **apdu);
```

Uruchomienie czytnika (z odpowiednimi parametrami), a następnie załączenie napięcia w odpowiednim slocie umożliwiają funkcje:

```
int scReaderInit(SC_READER_INFO *ri,
                 const char     *param);
int scReaderActivate(SC_READER_INFO *ri);
```

Zerowanie karty pozwala na uzupełnienie zawartości struktury SC_CARD_INFO. Po tej operacji możliwe jest przesyłanie komend do karty. Funkcjonalności tej dostarczają następujące funkcje:

```
int scReaderResetCard(SC_READER_INFO *ri,
                      SC_CARD_INFO   *ci);
int scReaderSendAPDU(SC_READER_INFO  *ri,
                     SC_CARD_INFO    *ci,
                     SC_APDU         *apdu);
```

Po zakończeniu korzystania z danego czytnika należy dokonać jego dezaktywacji (odcięcie napięcia od karty), a następnie wyłączyć go:

```
int scReaderDeactivate(SC_READER_INFO *ri);
int scReaderShutdown(SC_READER_INFO *ri);
```

Prosty przykład wykorzystania interfejsu dostarczanego przez bibliotekę *SCEZ* przedstawiono na wydruku 14. Program drukuje ATR karty oraz odpowiedź na przesyłaną komendę.

```
#include <stdlib.h>

// pliki nagłówkowe SCEZ
#include <scgeneral.h>
#include <screader.h>

int main(int argc, char *argv[])
{
    // informacje o czytniku
    SC_READER_INFO *ri;

    // informacje o karcie
    SC_CARD_INFO *ci;

    // konfiguracja czytnika
    SC_READER_CONFIG rc;

    // komenda i odpowiedź APDU
    SC_APDU *apdu;

    // komenda GET CHALLENGE
    BYTE GET_CHALLENGE[] = {0x00, 0x84, 0x00, 0x00, 0x08};

    // zmienna pomocnicza
    int ret, i;

    // inicjalizacja biblioteki
    scInit();

    // konfiguracja czytnika
    rc.type=SC_READER_TOWITOKO;
    rc.slot=1;
    rc.param="0";

    // inicjalizacja struktury ri
    ri=scGeneralNewReader(rc.type, rc.slot);

    // inicjalizacja struktury ci
    ci=scGeneralNewCard();

    // inicjalizacja czytnika
    scReaderInit(ri, rc.param);
```

```
    // aktywacja czytnika
    scReaderActivate(ri);

    // sprawdzenie obecności karty w slocie czytnika
    scReaderCardStatus(ri);
    if (!(ri->status&SC_CARD_STATUS_PRESENT))
    {
        printf("No card!\n");
        return 1;
    }

    // zerowanie karty
    scReaderResetCard(ri, ci);

    // wydruk ATR
    printf("ATR [%d] : ", ci->atrlen);
    for (i=0; i<ci->atrlen; i++)
        printf("%02X ", ci->atr[i]);

    printf("\n");

    // komenda APDU
    apdu=scGeneralNewAPDU();

    apdu->cmd=GET_CHALLENGE;
    apdu->cmdlen=sizeof(GET_CHALLENGE);
    apdu->cse=SC_APDU_CASE_2_SHORT;

    // przesłanie komendy do karty
    ret=scReaderSendAPDU(ri, ci, apdu);
    if (ret!=SC_EXIT_OK)
    {
        printf("Send APDU error!\n");
        return 1;
    }

    // wydruk odpowiedzi karty
    printf("RAPDU [%d] : ", apdu->rsplen);
    for (i=0; i<apdu->rsplen; i++)
        printf("%02X ", apdu->rsp[i]);
    printf("\n");

    // dezaktywacja czytnika
    scReaderDeactivate(ri);
    scReaderShutdown(ri);

    // zwolnienie pamięci
    scGeneralFreeCard(&ci);
    scGeneralFreeReader(&ri);
    scGeneralFreeAPDU(&apdu);

    // zakończenie działania biblioteki SCEZ
    scEnd();

    return 0;
}
```

Wydruk 14. Aplikacja wykorzystująca bibliotekę SCEZ (`SCEZExample.c`)

Oprócz możliwości wykorzystania czytników biblioteka *SCEZ* posiada również interfejs pozwalający wykryć rodzaj używanej karty. Zaimplementowane zostały również funkcje specyficzne dla pewnych typów kart, co ułatwia korzystanie z ich systemów operacyjnych.

6.7. Smart Card ToolKit

Przeznaczeniem tego pakietu jest możliwość komunikacji z kartami poprzez interfejs *Phoenix* (zobacz 5.2) lub *Smartmouse*, programowanie mikrokontrolerów PIC i AVR oraz zapis i odczyt danych z pamięci EEPROM z użyciem protokołu I2C.

Wraz z biblioteką dostarczone są proste programy konsolowe. Należy wspomnieć o aplikacji *7816shell*, który umożliwia komunikację poprzez interfejs *Phoenix* lub *Smartmouse* (zerowanie karty, przesyłanie i odbieranie danych). Program *avrspiprog* pozwala na programowanie mikrokontrolerów AVR.

6.7.1. Interfejs programisty

API jakie dostarcza biblioteka *Smart Card ToolKit* jest bardzo proste i pozwala jedynie na podstawowe operacje - otwarcie, zamknięcie urządzenia, zapis i odczyt danych, zerowanie.

W celu otwarcia urządzenia i utworzenia deskryptora połączenia należy użyć funkcji:

```
struct sc_desc_s *sctk_open(char                *ttysdev,
                            const enum sc_prot_e protocol);
```

ttysdev interfejs do którego podłączono urządzenie (np. `/dev/ttyS1`
protocol protokół komunikacyjny
- `sc_prot_phoenix` – użycie interfejsu *Phoenix*
- `sc_prot_smartmouse` – użycie interfejsu *Smartmouse*
- `sc_prot_i2c` – komunikacja z użyciem protokołu I2C
- `sc_prot_avrspi` – programowanie mikrokontrolerów AVR
- `sc_prot_picjdm` – programowanie mikrokontrolerów JDM

wartość zwracana wskaźnik do zainicjalizowanej struktury reprezentującej deskryptor połączenia

Przesłanie komendy związanej z używanym protokołem dokonywane jest poprzez:

```
int sctk_command(struct sc_desc_s *sc,
                 const int        command,
                 void             *data);
```

sc deskryptor połączenia
command komenda związana z protokołem
- `phs_command_direct_mode` – konwencja główna przesyłania znaków
- `phs_command_indir_mode` – konwencja odwrócona przesyłania znaków (odwrotna kolejność bitów niż w przypadku konwencji głównej)
- inne wartości – dla mikrokontrolerów AVR i PIC

data opcjonalne dane
wartość zwracana 0 jeśli operacja zakończyła się sukcesem

W celu ustawienia wartości czasów oczekiwania na określone zdarzenia oraz wykonywania pewnych operacji należy wykorzystać:

```
void sctk_setdelay(struct sc_desc_s      *sc,
                   const enum sc_delay_e id,
                   const unsigned int    delay);
```

sc deskryptor połączenia
id identyfikator modyfikowanej wartości
- `sc_delay_reset` – czas trwania sygnału zerowania karty

— `sc_delay_command` – czas oczekiwania na zakończenie komend zerowania karty oraz odczytu i zapisu danych
— `sc_delay_datatx` – czas oczekiwania pomiędzy przesłaniem do karty dwóch kolejnych bajtów
— `sc_delay_timeout` – czas oczekiwania na dane z karty

delay nowa wartość (w *ms*)
wartość zwracana brak

Przesłanie żądania zerowania karty realizuje funkcja:

```
int sctk_reset(struct sc_desc_s *sc);
```

sc deskryptor połączenia
wartość zwracana 0 jeśli operacja zakończyła się sukcesem

Przesłanie danych do karty realizowane jest za pomocą funkcji:

```
int sctk_write(struct sc_desc_s   *sc,
               const void         *data,
               const unsigned int size);
```

sc deskryptor połączenia
data bufor z danymi do przesłania
size ilość danych w buforze
wartość zwracana rzeczywista ilość przekazanych danych (w bajtach) lub -1 w przypadku błędu

Sprawdzenie ilości oczekujących danych (jakie zostały otrzymane od karty) można dokonać wykorzystując funkcję:

```
int sctk_ispending(struct sc_desc_s *sc);
```

sc deskryptor połączenia
wartość zwracana 0 w przypadku braku danych w buforze, wartość dodatnia jeśli istnieją dane do odczytu

Odczyt danych jakie pochodzą od karty możliwy jest przy użyciu:

```
int sctk_read(struct sc_desc_s   *sc,
              void               *data,
              const unsigned int size);
```

sc deskryptor połączenia
data bufor na dane
size wielkość bufora
wartość zwracana rzeczywista ilość odczytanych danych (w bajtach) lub -1 w przypadku błędu

Zamknięcie urządzenia i zwolnienie pamięci zajmowanej przez deskryptor możliwe jest przy użyciu:

```
void sctk_close(struct sc_desc_s *sc);
```

sc deskryptor połączenia
wartość zwracana brak

Pomocniczą funkcją, umożliwiającą opóźnienie aplikacji o określoną liczbę *ms*, jest:

```
void sctk_delay(unsigned int delay);
```

delay czas oczekiwania (w *ms*)
wartość zwracana brak

Przykład z wydruku 15 jest prostym programem zerującym kartę i przesyłającym komendę do karty (poprzez interfejs *Phoenix*).

```
// niezbędne pliki nagłówkowe
#include <stdio.h>
#include <stdlib.h>

// plik nagłówkowy biblioteki Smart Card ToolKit
#include "libsctk.h"

// wielkość bufora
#define MAX_BUFFER_LEN 255

int main(void)
{
    // deskryptor połączenia
    struct sc_desc_s *sc;

    // bufor na dane
    char buffer[MAX_BUFFER_LEN];
    unsigned int len;

    // komenda GET CHALLENGE
    char GET_CHALLENGE[] = {0x00, 0x84, 0x00, 0x00, 0x08};

    // otwarcie urządzenia podłączonego do COM1 (/dev/ttyS1)
    sc = sctk_open("/dev/ttyS1", sc_prot_phoenix);

    if (!sc)
    {
        fprintf(stderr, "Bla: nie mozna otworzyc urzadzenia.\n");
        return -1;
    }

    // ustawienie czasu oczekiwania na odczyt danych
    sctk_setdelay(sc, sc_delay_timeout, 200000);

    // ustawienie konwencji głównej przy przesyłaniu danych
    sctk_command(sc, phs_command_direct_mode, 0);

    // zerowanie karty
    sctk_reset(sc);

    // odczyt ATR
    memset(buffer, 0, MAX_BUFFER_LEN);
    len = sctk_read(sc, buffer, MAX_BUFFER_LEN);

    // brak odpowiedzi od karty
    if (len <= 0)
        printf("Brak ATR.\n");
    else
    {
        // odczytano ATR
        int i;
        printf("ATR [%i] =", len);
        for(i=0; i<len; ++i)
            printf(" %x", buffer[i]);
    }

    // przesłanie komendy do karty
    if (sctk_write(sc, GET_CHALLENGE, sizeof(GET_CHALLENGE))
            != sizeof(GET_CHALLENGE))
```

```
60          printf("Blad␣podczas␣przesylania␣danych.\n");
        else
            printf("Przeslano␣komende␣GET_CHALLENGE.\n");

        // odczyt odpowiedzi
65      memset(buffer, 0, MAX_BUFFER_LEN);
        if ((len = sctk_read(sc, buffer, MAX_BUFFER_LEN)) <= 0)
            printf("Brak␣odpowiedzi␣od␣karty.\n");
        else
        {
70          int i;
            printf("RAPDU␣[%i]␣=", len);
            for(i=0; i<len; ++i)
                printf("␣%x", buffer[i]);
        }
75
        // zamknięcie urządzenia
        sctk_close(sc);

        return 0;
80  }
```

Wydruk 15. Smart Card Tool Kit (`SCTKExample.c`)

Uwagi bibliograficzne

Implementacje interfejsów CT-API, PC/SC oraz liczne aplikacje i sterowniki dla systemu *Linux* można odnaleźć na stronach M.U.S.C.L.E. [102].

Zalecenia PC/SC opisują dokumenty [68, 69, 70, 71, 72, 73, 74, 75]. Dodatkowe informacje dotyczące programowania można odnaleźć w [80, 81].

Szczegółowe informacje o *OpenCT* znajdują się w podręczniku [83].

Interfejs programisty *Open Card Framweork* omawiają również publikacje [79, 78].

Biblioteka SATSA jest opisana szczegółowo w dokumentacji [89].

W [76] przedstawiono bibliotekę SCEZ.

Bibliotece SCTK poświęcona jest strona internetowa [106].

7. Realizacja aplikacji po stronie karty

Nieodłączną częścią systemu kartowego, oprócz aplikacji terminalowej, jest aplikacja stworzona na karcie. Może to być zarówno system plików przechowujących określone informacje, jak i kod wykonywany po stronie karty.

W rozdziale przedstawione są wybrane karty (ich możliwości, sposoby programowania), ogólne zalecenia dotyczące projektowania aplikacji kartowych oraz narzędzia wspomagające tworzenie tych aplikacji. W szczególności są to:

— narzędzia dla kart natywnych – większość producentów dostarcza wraz z określonym systemem operacyjnym oprogramowanie ułatwiające projektowanie aplikacji (zobacz 7.1),
— *Development Kit for the Java Card Platform* – jest to zestaw bibliotek i narzędzi umożliwiających tworzenie i testowanie aplikacji dla *Java Card* (zobacz 7.2),
— *GemXpresso RAD III* – narzędzie wspomagające projektowanie i testowanie aplikacji dla kart *Java* firmy *Gemplus* (zobacz 7.2.8),
— *Cyberflex Access Kits* – oprogramowanie przeznaczone dla kart firmy *Schlumberger* (zobacz 7.2.7).

7.1. Karty natywne

Projektowanie systemu wykorzystującego karty natywne wiąże się, w odniesieniu do samej karty, ze zdefiniowaniem systemu plików z danymi o odpowiednich prawach dostępu. Z tego powodu najważniejszym elementem realizacji takiej aplikacji jest jej szczegółowy projekt wraz z określeniem schematów użycia, modyfikacji i przepływu przechowywanych w karcie danych.

Pierwszym krokiem projektu aplikacji jest określenie zakresu jej funkcjonalności, czyli zadań do jakich jest przeznaczona. W początkowym etapie należy wyspecyfikować je na bardzo ogólnym stopniu szczegółowości (np. doładowanie portmonetki), a następnie starać się poszerzać poziom specyfikacji zwracając uwagę na funkcje, które dostępne są np. wyłącznie dla właściciela karty lub aplikacji. Dobrym sposobem postępowania jest opisanie wszystkich (lub prawie wszystkich) scenariuszy użycia. Pozwoli to mieć pewność, że nie pominiemy niczego w dalszym etapie projektu. Oprócz poprawnych schematów wykorzystania karty należy rozważyć również przypadki np. kilkukrotnego złego wprowadzenia kodu PIN (blokada karty bez możliwości odblokowania, zastosowanie dodatkowego kodu PUK, możliwość odblokowania karty przez wydawcę) lub błędnego uwierzytelnienia terminala.

W następnym kroku konieczne jest szczegółowe określenie zestawu danych przechowywanych przez kartę. Wiąże się to z: wyznaczeniem źródła danych, ich formatu, sposobu przechowywania oraz, co najważniejsze, praw dostępu do nich (zarówno jeśli chodzi o dostęp wyłącznie tylko do odczytu jak i związany z ich modyfikacją). Szczególnie należy zwrócić uwagę na kody PIN oraz klucze kryptograficzne. Brak precyzyjnego określenia na tym etapie jakie klucze związane są z uwierzytelnieniem użytkownika, kto jest odpowiedzialny za ich generację, jakie schematy operacji kryptograficznych są obowiązujące doprowadzi do kłopotów na etapie fizycznej realizacji aplikacji lub jej wdrażania. Już na tym etapie trzeba pomyśleć o bezpieczeństwie związanym z zarządzaniem kluczami oraz ich wykorzystaniem. Stosowane algorytmy kryptograficzne powinny być odpowiednio silne, a ilość kluczy odpowiednia do ilości danych i sposobu operowania na nich. Pozwoli to na zmniejszenie prawdopodobieństwa kompromitacji całego systemu w przypadku złamania lub wycieku jednego z kluczy.

Kolejny etap związany jest już ściśle z możliwościami systemu operacyjnego jaki dostarcza dana karta. Określone wcześniej dane należy podzielić ze względu na ich znaczenie, sposoby operowania nimi oraz prawa dostępu na katalogi i pliki. Stworzenie struktury na samej karcie jest pierwszym elementem tworzenia skryptu personalizującego karty. Należy zwrócić uwagę na

odpowiednie rozmieszczenie danych na karcie, tak aby zminimalizować liczbę wykonywanych operacji podczas sesji komunikacyjnej.

Po utworzeniu kilku kart można przystąpić do testowania funkcjonalności jakie w założeniu mają spełniać. Takie testy są jednocześnie przełożeniem scenariuszy użycia na komendy APDU. Oczywiście wykonywane scenariusze nie mogą ograniczać się wyłącznie do poprawnych przebiegów komunikacji. Należy prześledzić wszystkie możliwe ścieżki wykonania komend w przypadku np. podania nieprawidłowego kodu PIN lub nawiązania niepoprawnej komunikacji w trybie zabezpieczonym (ang. *secure messaging*). W przypadku wykrycia niepożądanych zachowań należy zmodyfikować prawa dostępu lub rozkład danych. Warto dokumentować przeprowadzone testy, gdyż posłużą one do implementacji aplikacji terminalowej. W przyszłości, po wprowadzeniu zmian do systemu, dokumentacja przeprowadzonych testów może posłużyć do walidacji nowej implementacji.

Wraz z określonym systemem operacyjnym dla kart producenci zazwyczaj dostarczają pomocnicze narzędzia ułatwiające projektowanie aplikacji. Przykładem mogą być programy z serii *Pilot* firmy *Gemplus*. Posiadają one prosty interfejs umożliwiający przesyłanie dowolnych komend APDU do kart (nie ma znaczenia z jakim systemem operacyjnym), tworzenie skryptów (automatyczne wykonanie zestawu komend) oraz wysokopoziomowe zestawy operacji pozwalające, dla danej karty, korzystać z jej specyficznych możliwości (tworzenie plików elektronicznej portmonetki, obsługa programu lojalnościowego). Dla tych operacji komendy APDU generowane są automatycznie, co ułatwia projektowanie aplikacji terminalowej.

Wspomaganie projektowania aplikacji w architekturze PKCS #15 możliwe jest z użyciem narzędzi *OpenSC* (zobacz 10.2.1). Zawarte w pakiecie oprogramowanie pozwala na personalizację kart i ich wykorzystanie do operacji charakterystycznych dla infrastruktury klucza publicznego.

W każdym z przypadków projektowania aplikacji kartowych warto stworzyć proste oprogramowanie będące symulatorem systemu zewnętrznego dla karty. W większości stworzony kod wykorzystany zostanie przy implementacji docelowej aplikacji kartowej dla terminala. Oprogramowanie pozwoli również na wszechstronne testy.

7.2. Aplety dla Java Card

Implementacja wirtualnej maszyny języka Java (JCVM, ang. *Java Card Virtual Machine*) na karcie inteligentnej otworzyła zupełnie nowe możliwości programowania i wykorzystania kart. Poprzez możliwość umieszczania na karcie apletów o różnorodnym przeznaczeniu, nie ma już ograniczeń wynikających z funkcjonalności systemu operacyjnego. Środowisko na które składa się JCVM, *Java Card* API oraz funkcje pomocnicze nosi nazwę JCRE (ang. *Java Card Runtime Enviroment*).

Poniżej opisano środowisko do tworzenia oraz emulowania działania apletów kartowych rozwijane przez firmę *Sun Microsystems*. Na jego przykładzie przedstawione są etapy rozwoju aplikacji oraz architektura *Java Card*. W dalszej części czytelnik zapozna się z dedykowanymi narzędziami dla rzeczywistych kart na przykładzie *Cyberflex* oraz *GemXpresso*.

7.2.1. Instalacja oprogramowania

Development Kit for the Java Card Platform zawiera zestaw narzędzi pozwalający zapoznać się z technologią kart zawierających maszynę wirtualną Java. Rozwijany jest w wersji binarnej dla platform *Linux*, *Sun Solaris* oraz *Microsoft Windows*. Pełną funkcjonalność otrzymamy instalując następujące pakiety:

— *Java 2 Platform, Standard Edition* – zawiera podstawowe składniki języka Java (kompilator, pakiety itp.) [10],
— *OpenCard Framework* – pakiet przeznaczony do obsługi czytników kart inteligentnych z poziomu aplikacji w języku Java (zobacz również 6.4),
— *Development Kit for the Java Card Platform* – zawiera pakiety i oprogramowanie pozwalające tworzyć, kompilować i testować aplety kartowe [11].

Instalacja oprogramowania polega na rozpakowaniu do wybranych katalogów archiwów w formacie *zip*. Do prawidłowego działania niezbędne jest prawidłowe ustawienie następujących zmiennych środowiskowych:

— `JC_HOME` – wskazuje na katalog, w którym umieszczone zostały pliki *Development Kit for the Java Card Platform*,
— `JAVA_HOME` – wskazuje na katalog, w jakim zainstalowano *Java 2 Platform, Standard Edition*,
— `PATH` – należy uzupełnić o następujące ścieżki: JC_HOME/bin oraz JAVA_HOME/bin (pozwoli to na uruchamianie oprogramowania bez konieczności podawania pełnej ścieżki).

Podczas instalacji należy zwrócić uwagę na ścieżki do docelowych katalogów. Jeżeli zawieriają one znak spacji mogą wystąpić problemy podczas uruchamiania skryptów wywołujących odpowiednie programy. Niezbędna jest wtedy modyfikacja skryptów startowych polegająca na ujęciu w znaki cytowania pełnych ścieżek do pakietów.

7.2.2. Architektura Java Card

Podczas produkcji *Java Card* pamięci ROM umieszczona zostaje JCVM (ang. *Java Card Virtual Machine*) oraz niezbędne pakiety i biblioteki. Istnieje jednak zasadnicza różnica pomiędzy kartową a „tradycyjną” (uruchamianą na komputerze klasy PC) maszyną języka Java. JVM (ang. *Java Virtual Machine*) jest zwykłym procesem, który po zakończeniu jest usuwany z pamięci komputera, natomiast JCVM nawet po odcięciu zasilania od karty pozostaje w pamięci. Utworzone obiekty nie są niszczone, po ponownym umieszczeniu karty w czytniku są odzyskiwane. Można powiedzieć, że JCVM jest ciągłym procesem zawieszanym na czas gdy karta jest odłączona od zasilania.

Java Cards wyposażone są zazwyczaj w 8- lub 16-bitowy procesor. Minimalna ilość pamięci nieulotnej wynosi 16 kB. Bardziej zaawansowane karty posiadają 32-bitowy procesor oraz koprocesor kryptograficzny.

Aplikacje na *Java Card* tworzone są w postaci apletów kartowych. Po jego utworzeniu, kompilacji, przetworzeniu na specjalny format i przetestowaniu zostają umieszczone na karcie. Tu aplet będzie zarejestrowany przez JCRE co oznacza utworzenie jego instancji. Stworzone przez aplet obiekty istnieją póki aplet nie zostanie usunięty z karty. Do komunikacji i obsługi apletu przez JCRE używane są następujące metody:

— *install* – stworzenie przez aplet obiektów i ich inicjalizacja (po umieszczeniu apletu na karcie); aplet jest również rejestrowany na karcie (posiada unikalne AID),
— *select* – wybór apletu do którego mają być przesyłane komendy; aktywacja apletu powoduje dezaktywację apletu poprzednio wybranego (wyjątkiem od tej reguły jest wykorzystanie kanałów logicznych),
— *deselect* – dezaktywacja aktualnie wybranego apletu (ma on możliwość wykonania pewnych operacji kończących aktualną sesję),
— *process* – przesłanie do apletu komendy APDU i jej obsłużenie,
— *uninstall* – wywołanie tej metody poprzedza usunięcie apletu z karty.

[10] `http://java.sun.com/j2se`
[11] `http://java.sun.com/products/javacard`

Wymienione warianty komunikacji maszyny wirtualnej z apletem przekładają się na publiczne metody zaimplementowane w aplecie (zobacz wydruk 16).

```
// szkielet apletu kartowego

// nazwa pakietu
package pl.edu.pw.ii.scproj.samples.AppletFrame;
// imprort klas z pakietu javacard.framework
import javacard.framework.*;
// klasa AppletFrame
public class AppletFrame extends Applet implements AppletEvent
{
    // konstruktor
    protected AppletFrame()
    {
        // rejestracja apletu
        register();
    }

    // metoda wywoływana po umieszczeniu nowego apletu na karcie
    public static void install(byte[] bArray, short bOffset, byte bLength)
    {
        // stworzenie instancji apletu na karcie
        new AppletFrame();
    }

    // metoda wywoływana podczas aktywacji apletu, jej implementacja nie jest obowiązkowa
    public void select()
    {
        // czynności wykonywane na początku sesji apletu
    }

    // metoda wywoływana w celu obsłużenia przez aplet komendy APDU
    public void process(APDU apdu) throws ISOException
    {
        byte buffer[] = apdu.getBuffer();
        // obsłużenie komendy
        // ...
    }

    // metoda wywoływana podczas dezaktywacji apletu, jej implementacja nie jest obowiązkowa
    public void deselect()
    {
        // czynności wykonywane na koniec sesji apletu
    }

    // metoda wywoływana przed usunięciem instancji apletu z karty,
    // jej implementacja nie jest obowiązkowa
    public void uninstall()
    {
        // czynności wykonywane przed usunięciem instancji apletu z karty
    }
}
```

Wydruk 16. Szkielet apletu (`AppletFrame.java`)

JCVM posiada szereg ograniczeń w porównaniu do JVM wynikających z niewielkich zasobów na karcie. Do najważniejszych należą:

— brak typów danych: *long*, *double*, *float*, *char*,
— brak klasy **java.lang.System** (została zastąpiona przez **javacard.framework.JCSystem**),
— brak procesu *garbage collection* – nie ma możliwości dynamicznego usuwania obiektów (ponadto w rzeczywistych kartach nie ma możliwości defragmentacji użytej pamięci),
— brak *Security Manager*,
— nie ma możliwości stosowania wątków, dynamicznego ładowania klas, klonowania obiektów oraz używania wielowymiarowych struktur danych,

— brak wsparcia dla metody **finalize()** oraz słów kluczowych *synchronized*, *transient*, *native*, *volatile*, *strictfp*, a także wielu klas z tradycyjnej implementacji Java np. **String**.

Opcjonalnie wspierane są: możliwość stosowania typu *int* oraz usuwania obiektów. Pozostałe cechy języka Java takie jak: mechanizm wyjątków, dziedziczenie, kontrola rzutowania typów znajdują odzwierciedlenie w implementacjach aplikacji na karcie.

7.2.3. Java Card API

Podczas tworzenia aplikacji dla kart najczęściej korzysta się z następujących pakietów języka Java:

java.lang zawiera podzbiór podstawowych klas języka Java przeznaczonych dla kart,

java.io wykorzystujemy jedynie podzbiór tego pakietu związany z wyjątkami,

java.rmi pakiet używany do definiowania zdalnego interfejsu karty czyli metod wywoływanych przez CAD (ang. *chip accepting device*); pozwala to na tworzenie aplikacji terminalowej w architekturze RMI (ang. *Remote Method Invocation*),

javacard.framework dostarcza strukturę klas i interfejsów używanych przy tworzeniu i komunikowaniu się z apletami kartowymi,

javacard.security zestaw klas i interfejsów związanych z bezpieczeństwem np. klasy reprezentujące klucze kryptograficzne,

javacardx.crypto zestaw dodatkowych funkcji kryptograficznych.

Wszystkie klasy wywodzą się od podstawowej klasy języka Java **java.lang.Object**. Klasa ta jest znacznie uproszczona w porównaniu do klasy **Object** z tradycyjnego pakietu języka Java. Zawarto w niej wyłącznie metodę służącą do porównywania obiektów. Nie ma możliwości ich klonowania. Poniżej przedstawiono najczęściej wykorzystywane klasy wraz z krótkim opisem. Przykłady ich zastosowania odnajdzie czytelnik w demonstracyjnych aplikacjach (zobacz np.: 9.2, 8.2.4).

javacard.framework.Applet jest to abstrakcyjna klasa bazowa dla wszystkich apletów,

javacard.framework.JCSystem zawiera szereg statycznych metod przeznaczonych do obsługi wykonywanego apletu, zarządzania zasobami, współdzielenia obiektów pomiędzy aplikacjami, usuwania obiektów oraz realizacji transakcji atomowych,

javacard.framework.AID używana jest do przechowywania AID (ang. *application identifier*) aplikacji; JCRE tworzy instancję tej klasy w celu zarządzania i identyfikacji apletu, sam aplet może używać metod **JCSystem.getAID()** (w celu pobrania własnego AID) oraz **JCSystem.lookupAID()** (w celu pobrania AID innego apletu),

javacard.framework.APDU wykorzystywana do przechowywania rozkazu albo odpowiedzi APDU; jest to tymczasowy obiekt JCRE *Entry Point Object*, co oznacza, że nie można tworzyć zmiennych referencyjnych z nim związanych; klasa zaprojektowana jest do obsługi APDU niezależnie od stosowanego protokołu,

javacard.framework.Util klasa zawiera zestaw przydatnych statycznych metod (kopiowanie i porównywanie tablic, konkatenacja i rozdzielanie bajtów),

javacard.framework.OwnerPIN klasa przeznaczona do przechowywania i zarządzania kodem PIN właściciela karty (implementuje interfejs **javacard.framework.PIN**); zawiera mechanizmy chroniące kod PIN takie jak licznik niepoprawnych wprowadzeń oraz mechanizm blokujący.

Do często używanych klas należą również obiekty reprezentujące wyjątki np.: zły format komendy APDU, zły typ danych, niepoprawna operacja.

Najważniejszymi interfejsami dostarczanymi wraz z *Development Kit for the Java Card Platform* są:

javacard.framework.ISO7816 zawiera zestaw predefiniowanych stałych, które są związane z normami ISO/IEC 7816-3 oraz ISO/IEC 7816-4,

javacard.security.Key bazowy interfejs dla wszystkich klas reprezentujących klucze,

javacard.framework.PIN interfejs reprezentuje kod PIN wraz z mechanizmami do zarządzania i ochrony jego wartości,

java.rmi.Remote służy do identyfikacji interfejsów, których metody mogą być wywoływane zdalnie poprzez CAD (ang. *chip accepting device*).

7.2.4. Rozwój aplikacji

Poniżej przedstawione są kolejne czynności jakie należy wykonać by stworzyć aplet, umieścić go na karcie, stworzyć jego instancję, a następnie usunąć.

Implementacja

Pierwszym krokiem zmierzającym do implementacji apletu kartowego jest określenie jego funkcjonalności. Należy sprecyzować jakie aplet ma wykonywać zadania i w jakich warunkach. Ważne jest również określenie możliwych scenariuszy przebiegu poszczególnych funkcji. Następnie należy zaprojektować strukturę klas (pakiet) apletu. W tym kroku konieczne jest również przydzielenie identyfikatora AID zarówno dla pakietu jak i klasy bezpośrednio dziedziczonej po klasie **Applet**. Można teraz przystąpić do zaprojektowania interfejsu pomiędzy apletem a środowiskiem zewnętrznym. Wymaga to zdefiniowania komend APDU służących do realizacji funkcji apletu. Dla każdej komendy należy wyznaczyć wszystkie możliwe warianty odpowiedzi. Zadanie to będzie ułatwione, jeżeli dobrze określono wymagania funkcjonalne.

Po implementacji apletu (w formie pliku źródłowego języka Java) można przystąpić do jego testowania (czyli sprawdzenia czy zachowuje się i działa zgodnie z naszymi założeniami). *Java Card Development Kit* zawiera szereg programów przeznaczonych do tego celu. Poniżej przedstawiono kolejne kroki jakie należy wykonać, aby skompilować, skonwertować, przetestować, umieścić na karcie a następnie z niej usunąć aplet.

Kompilacja plików źródłowych

Pliki źródłowe (**.java*) kompilujemy przy użyciu kompilatora *javac*, koniecznie z opcją *-g*. Należy zwrócić uwagę na podanie ścieżki do potrzebnych klas. W wyniku kompilacji otrzymujemy pliki typu **.class*.

Konwersja plików **.class*

Przed konwersją plików **.class* należy utworzyć plik z parametrami. Przykład takiego pliku przedstawiono na wydruku 17.

```
1 out EXP JCA CAP
2 -exportpath .
3 -applet 0xa0:0x0:0x0:0x0:0x62:0x0:0x0:0x0:0x0:0x2:0x1 \
4     pl.edu.pw.ii.scproj.samples.AppletFrame.AppletFrame
5 pl.edu.pw.ii.scproj.samples.AppletFrame
6 0xa0:0x0:0x0:0x0:0x62:0x0:0x0:0x0:0x0:0x2 1.0
```

Wydruk 17. Plik z opcjami konwersji apletu (`AppletFrame.opt`)

Kolejne linie pliku *AppletFrame.opt* zawierają opcje dla programu *converter*. Opcje te można podać również jawnie podczas uruchomienia programu. Wygodniejsze jest jednak stworzenie pliku zawierającego wartości odpowiednich parametrów.

Pierwsza linia zawiera rodzaje plików jakie ma wygenerować narzędzie. W kolejnych zawarte są: ścieżka dostępu do plików eksportu pakietów używanych przez aplet (linia druga),

nazwa apletu wraz z numerem AID (składającego się z RID[12] oraz PIX, linia trzecia i czwarta), nazwę pakietu (linia piąta) oraz jego AID wraz z numerem wersji (linia szósta). Kolejność oraz ilość opcji w pliku może być inna.

Po stworzeniu pliku z parametrami możemy uruchomić narzędzie *converter* z opcją *-config <nazwa_pliku_opt>*.

W wyniku jego działania otrzymamy pliki z następującymi rozszerzeniami:

— *CAP* – jest to plik w formacie *JAR* zawierający wykonywalną wersję klas w pakiecie; plik jest zapisany w formie binarnej, a jego tekstową wersję można uzyskać korzystając z narzędzia *capdump*; plik CAP składa się z wielu komponentów zawierających klasy, metody itp.,
— *JCA* (ang. *Java Card Assembly*) – plik tekstowy zawierający aplet przekonwertowany na format asemblera *Java Card*,
— *EXP* – plik eksportu dla pakietu; jest to plik binarny, a jego formę tekstową można otrzymać korzystając z narzędzia *exp2text*.

Z plików *JCA* można wygenerować pliki *CAP* używając narzędzia *capgen*. W przypadku wystąpienia błędu informującego o złym formacie plików **.class* należy powtórzyć etap kompilacji z opcją *-target 1.1*[13].

Weryfikacja

Kolejnym krokiem jest weryfikacja otrzymanych plików wynikowych pod względem:

— weryfikacji zależności i zgodności wersji pomiędzy plikiem *CAP* a plikami *EXP* – służy do tego narzędzie *verifycap*,
— weryfikacji pojedynczych plików *EXP* – narzędziem używanym do tego celu jest *verifyexp*,
— weryfikacji zgodności wersji pomiędzy plikami *EXP* – dedykowane narzędzie nosi nazwę *verifyrev*.

Weryfikacja poprawności pozwala uniknąć sytuacji nieoczekiwanego zachowania apletu przed umieszczeniem go na karcie.

Emulacja działania apletu

Środowisko JCWDE (ang. *Java Card Workstation Development Environment*) pozwala na emulowane wykonanie stworzonego apletu[14]. Przed uruchomieniem symulatora należy przygotować jego plik konfiguracyjny. Zawarte są w nim nazwy apletów oraz ich numery AID, które chcemy testować. Przykład takiego pliku pokazano na wydruku 18.

```
1 // nazwy apletów oraz ich AID
2 pl.edu.pw.ii.scproj.samples.AppletFrame 0xa0:0x0:0x0:0x0:0x0:0x62:0x0:0x0:0x0:0x0:0x2
3 com.sun.javacard.samples.wallet.Wallet 0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
```

Wydruk 18. Plik konfiguracyjny dla narzędzia JCWDE (`jcwde.cfg`)

Po uruchomieniu *jcwde* z wyspecyfikowanym plikiem konfiguracyjnym nasłuchuje on na porcie 9025. Komendy możemy przesyłać w formacie T=0 przy użyciu narzędzia *apdutool*. Najwygodniej przesyłane komendy (liczby w formacie heksadecymalnym) wpisać do pliku.

Dużo bardziej rozbudowanym narzędziem jest *cref*. Pozwala on na pełną emulację zachowania *Java Card* łącznie z możliwością tworzenia instancji, usuwania apletów, używania kanałów logicznych oraz zerowania karty. Program ten zachowuje zawartość pamięci EEPROM

[12] w prezentowanych przykładach został użyty RID z przykładowych apletów firmy *Sun Microsystems* (0x0a00000062); w przypadku rozwoju własnych aplikacji należy wykupić własny numer RID; numery te są kontrolowane i zarządzane przez ISO; rejestracja własnego RID jest płatna

[13] autor kompilował aplety z użyciem kompilatora w wersji 1.4.2

[14] jest to emulacja wyłącznie samego apletu, a nie *Java Card*; nie jest więc możliwe ładowanie oraz usuwanie apletów i inne operacje typowe dla zarządzania samą kartą

w plikach na dysku. Pozwala to na wielokrotne użycie tej samej wirtualnej karty. Z *cref* komunikujemy się podobnie jak z *jcwde*. Pomocnicze narzędzie *scriptgen* pozwala na automatyczną konwersję plików *CAP* na komendy APDU w formacie czytelnym dla programu *apdutool*. Program *cref* symuluje zachowanie apletu zarządzającego pakietami i innymi apletami na karcie. Zestaw komend, które pozwalają na operacje zmieniające zawartość i aktualnie wykonywany aplet jest następujący (parametr q przyjmujący wartości 0, 1 lub 2 pozwala na komunikację określonym kanałem logicznym):

— **SELECT FILE** – umożliwia wybór apletu na karcie (w opisie podano kolejno <*CLA*> <*INS*> <*P1*> <*P2*> oraz opcjonalne dane)

```
0x0q 0xa4 0x04 0x00 <LC> <AID> 0x7f
```

LC długość pola AID
AID identyfikator AID apletu

— **CAP BEGIN** – komenda ta jest informacją dla apletu zarządzającego, że kolejne instrukcje będą zawierały kolejne komponenty pliku CAP

```
0x8q 0xb0 0x00 0x00 0x00 0x7f
```

— **CAP END** – oznacza zakończenie transmisji komponentów pliku CAP

```
0x8q 0xba 0x00 0x00 0x00 0x7f
```

— **COMPONENT ## BEGIN** – rozpoczęcie transmisji komponentu pliku CAP o numerze sekwencyjnym ##

```
0x8q 0xb2 0x## 0x00 0x00 0x7f
```

— **COMPONENT ## END** – zakończenie transmisji komponentu pliku CAP o numerze sekwencyjnym ##

```
0x8q 0xbc 0x## 0x00 0x00 0x7f
```

— **COMPONENT ## DATA** – przesłanie danych komponentu ##

```
0x8q 0xb4 0x## <LC> <DATA> 0x7f
```

LC długość pola DATA
DATA dane określonego komponentu

— **CREATE APPLET** – utworzenie instancji apletu na karcie

```
0x8q 0xb8 0x00 0x00 <LC> <LCAID> <AID> <LCPAR> <PAR> 0x7f
```

LC długość pól AID oraz PAR (łącznie z polami określającymi ich długość)
LCAID długość pola AID
AID identyfikator AID apletu, którego instancję chcemy utworzyć
LCPAR długość pola PAR
PAR parametry przekazywane do apletu

— **ABORT** – przerwanie transmisji pliku CAP (przesłane już dane zostaną utracone)

```
0x8q 0xbe 0x00 0x00 0x00 0x7f
```

— **DELETE PACKAGE** – usunięcie pakietu z karty

```
0x8q 0xc0 <P1> <P2> <LC> <LCAID> <AID> 0x7f
```

P1, P2 dowolne wartości (ignorowane)
LC sumaryczna długość pół LCAID i AID
LCAID długość pola AID

AID identyfikator AID usuwanego pakietu

— **DELETE PACKAGE AND APPLETS** – jednoczesne usunięcie pakietu i apletów z karty

```
0x8q 0xc2 <P1> <P2> <LC> <LCAID> <AID> 0x7f
```

P1, P2 dowolne wartości (ignorowane)
LC sumaryczna długość pól LCAID i AID
LCAID długość pola AID
AID identyfikator AID usuwanego pakietu i apletów

— **DELETE APPLETS** – usunięcie instancji apletów z karty o określonych AID

```
0x8q 0xc4 <P1> <P2> <LC> <LCAIDn> <AIDn> ... 0x7f
```

P1 ilość usuwanych apletów (wartości od 0x01 do 0x08)
P2 dowolna wartość (ignorowane)
LC sumaryczna długość wszystkich par pól LCAIDn i AIDn
LCAIDn długość kolejnego identyfikatora AIDn apletu
AIDn AID apletu do usunięcia

Szczegółowy opis formatu instrukcji oraz kodów powrotu zawarty jest w [96].

Jak już wspomniano program *apdutool* pozwala komunikować się z emulatorami. Dla programu należy przygotować skrypt z kolejnymi komendami przesyłanymi do karty oraz wpływającymi na generowane komunikaty i zachowanie czytnika:

— **powerup** – włączenie czytnika; komendę tą należy umieścić na początku skryptu (przed komendami APDU),
— **powerdown** – wyłączenie czytnika,
— **delay** <*liczba*> – zatrzymuje wykonanie apletu przez określony czas,
— **echo** ”*ciąg znaków*”**”** – wydruk tekstu do pliku z wynikami,
— **output on** – włącza wydruk wyników działania,
— **output off** – wyłącza wydruk wyników działania.

Przykładowy skrypt dla *apdutool* zaprezentowano na wydruku 19. Plik wynikowy zawiera wysłane komendy oraz odpowiedzi na te pliki.

```
// włączenie czytnika
powerup;
// przesłanie komendy w postaci:
// <CLA> <INS> <P1> <P2> <LC> [<bajt 0> <bajt 1> ... <bajt LC-1>] <LE>;
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
// 90 00 = SW_NO_ERROR (oczekiwana odpowiedź od karty)
// wyłączenie drukowania wyników (odpowiedzi karty)
// np. podczas ładowania kardletów na kartę
output off;
// ...
// włączenie drukowania wyników
output on;
// ...
// wyłączenie czytnika
powerdown;
```

Wydruk 19. Przykład skryptu z komendami APDU (`capdu.scr`)

Narzędzie *scriptgen* konwertuje jedynie pliki *CAP* do postaci czytelnej dla programu *apdutool*. Aby umieścić aplet na karcie trzeba najpierw wybrać aplet umożliwiający wgranie pliku *CAP* (przy użyciu zestawu komend apletu zarządzającego). Po jego poprawnym umieszczeniu możemy stworzyć instancję apletu. Pełny przykład operacji wykonywanych z użyciem narzędzia *cref* wraz z komentarzem pokazano na wydruku 20.

```
// włączenie czytnika
powerup;
```

```
// wybranie apletu obsługującego instalację o AID
// 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01
0x00 0xa4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7f;

// komendy wygenerowane przez narzędzie scriptgen dla danego pliku (plików) CAP
// rozpoczęcie transmisji pliku cap
0x80 0xb0 0x00 0x00 0x00 0x7f;

// rozpoczęcie transmisji komponentu 0x01
0x80 0xb2 0x01 0x00 0x00 0x7f;
// dane komponentu
0x80 0xb4 0x01 0x00 ...
// zakończenie transmisji komponentu 0x01
0x80 0xbc 0x01 0x00 0x00 0x7f;

// rozpoczęcie transmisji komponentu 0x02
0x80 0xb2 0x02 0x00 0x00 0x7f;
// dane komponentu
0x80 0xb4 0x02 0x00 ...
// zakończenie transmisji komponentu 0x02
0x80 0xbc 0x02 0x00 0x00 0x7f;

// zakończenie transmisji pliku CAP
0x80 0xba 0x00 0x00 0x00 0x7f;

// transmisja kolejnych plików CAP
// ...

// stworzenie instancji apletu A o AID
// 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x02 0x01
0x80 0xb8 0x00 0x00 0x0c 0x0a 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x02
0x01 0x00 0x7f;

// stworzenie instancji apletu A z nowym AID
// 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x02 0x01 0x02
0x80 0xb8 0x00 0x00 0x1b
0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x02 0x01
0x0b 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x02 0x01 0x02
0x7f;

// wybranie apletu obsługującego instalację
0x00 0xa4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7f;

// usunięcie apletu o AID
// 0x0b 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x07 0x01 0x01
0x80 0xc4 0x01 0x00 0x0c 0x0b 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x07
0x01 0x01 0x7f;

// usunięcie pakietu o AID
// 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x07 0x03
0x80 0xc0 0x00 0x00 0x0b 0x0a 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x0c 0x07 0x03 0x7f;

// wyłączenie czytnika
powerdown;
```

Wydruk 20. Umieszczanie, tworzenie instancji i usuwanie apletu (`capduf.scr`)

7.2.5. Przykłady implementacji

Poniżej są przedstawione, w formie przykładów, typowe problemy implementacyjne z jakimi zetknie się programista *Java Card*. Pełne aplikacje są zaprezentowane w rozdziałach dotyczących konkretnych zagadnień kartowych (elektroniczna portmonetka, lojalność).

Obsługa APDU

Obsługa przesłanej przez użytkownika komendy APDU wiąże się z odpowiednim parsowaniem komendy oraz przesłaniem odpowiedzi. Dane w odpowiedzi umieszczane są w buforze, natomiast błędy sygnalizowane są przez mechanizm wyjątków, które przekładają się na odpowiednie, dwubajtowe, kody powrotu.

```
// klasy oraz instrukcje APDU warto zdefiniować
final static byte SAMPLE_CLA=(byte)0x80;
final static byte SAMPLE_INS=(byte)0x20;

// metoda przetwarzająca komendę APDU
public void process(APDU apdu)
{
    // pobranie zawartości bufora komendy
    byte[] buffer = apdu.getBuffer();

    // w przypadku użycia kanałów logicznych konieczne jest użycie maski,
    // która spowoduje zignorowanie dwóch najmłodszych bitów (oznaczających
    // użyty kanał logiczny)
    buffer[ISO7816.OFFSET_CLA]=(byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);

    // sprawdzenie czy klasa komendy jest prawidłowa
    if (buffer[ISO7816.OFFSET_CLA]==SAMPLE_CLA)
    {
        switch (buffer[ISO7816.OFFSET_INS])
        {
            // podjęcie odpowiednich działań (w zależności od instrukcji)
            case SAMPLE_CLA :
                // wysłanie w polu danych 0x0102
                Util.setShort(buffer, (short)0, (short)0x0102);
                apdu.setOutgoingAndSend((short)0, (short)2);
                break;
            // obsługa kolejnych instrukcji
            case ... :
                // ...
                break;
            default:
                // wyjątek - instrukcja nieobsługiwana
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
    // jest to komenda ISO 7816
    else if (buffer[ISO7816.OFFSET_CLA]==ISO7816.CLA_ISO7816)
    {
        // instrukcja SELECT FILE
        if (buffer[ISO7816.OFFSET_INS]==ISO7816.INS_SELECT)
        {
            if (selectingApplet())
            {
                // była to komenda dotycząca apletu
            }
            else
            {
                // komenda dotyczyła wewnętrznych obiektów apletu
            }
        }
        // wyjątek - instrukcja nieobsługiwana
        else ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
    // wyjątek - klasa instrukcji nieobsługiwana
    else ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
}
```

Wydruk 21. Obsługa APDU (`APDU.java`)

Transakcje

Mechanizm transakcji pozwala na zapewnienie wykonania ciągu określonych instrukcji w całości albo (w razie awarii, zgłoszenia wyjątku lub błędu karty) niewykonanie żadnej z nich.

```
try
{
    // rozpoczęcie transakcji
    JCSystem.beginTransaction();
    // wykonanie szeregu operacji
    // ...
    // zatwierdzenie transakcji
    JCSystem.commitTransaction();
}
// w przypadku zaistnienia wyjątku
catch (Exception e)
{
    // cofnięcie transakcji
    JCSystem.abortTransaction();
}
```

Wydruk 22. Transakcje (`Transaction.java`)

Obiekty tymczasowe

Obiekty tymczasowe (ang. *transient objects*) istnieją jedynie przez określony czas. Po zakończeniu wykonywania apletu (wybranie innego apletu albo ponowne zerowanie karty) ich wartość jest kasowana.

```
// deklaracje tymczasowych obiektów
private short[] transientShorts;
private boolean[] transientBools;

// konstruktor apletu
protected TransientObj(...)
{
    // ...
    try
    {
        // obiekt tymczasowy - 2 liczby typu short tracone w wypadku zmiany
        // apletu lub zerowania karty
        transientShorts=JCSystem.makeTransientShortArray(2,
                                                JCSystem.CLEAR_ON_DESELECT);

        // obiekt tymczasowy - 4 wartości logiczne tracone w wypadku zerowania
        // karty
        transientBools=JCSystem.makeTransientBooleanArray(4,
                                                JCSystem.CLEAR_ON_RESET);
        // ...
    }
    catch (NegativeArraySizeException e)
    {
        // ujemna wartość wielkości tablicy obiektów
    }
    catch (SystemException e)
    {
        // błąd systemowy - brak pamięci
    }
}

protected SampleMethod(...)
{
    // sprawdzenie czy obiekt jest tymczasowy
    if (JCSystem.isTransient(transientBools))
    {
        // obiekt jest tymczasowy
        // ...
```

```
        }
40      // ...
}
```

Wydruk 23. Obiekty tymczasowe (`TransientObj.java`)

Tworzenie i usuwanie obiektów

Mimo iż *Java Cards* nie posiadają mechanizmu *garbage collection* można zwolnić pamięć w przypadku utraty referencji do obiektów. Służy temu specjalna metoda klasy **JCSystem**. Przez utratę referencji rozumie się jej brak jako zmiennej statycznej lub obiektu w klasie. Nie należy nadużywać tej metody programowania, ponieważ liczba zapisów do pamięci EEPROM jest ograniczona. Nie są to także szybkie operacje.

```
void changeBuffer(byte newSize)
{
    if(buffer!=null && buffer.length==newSize)
        // nie ma potrzeby zmiany wielkości bufora
5       return;
    try
    {
        // rozpoczęcie transakcji
        JCSystem.beginTransaction();
10      byte[] oldBuffer = buffer;
        // utworzenie nowego bufora
        buffer = new byte[requiredSize];
        // jeśli istniał bufor
        if (oldBuffer!=null)
15          // żądanie usunięcia obiektów z pamięci
            JCSystem.requestObjectDeletion();
        // zatwierdzenie transakcji
        JCSystem.commitTransaction();
    }
20  catch (Exception e)
    {
        // odwołanie transakcji w przypadku wyjątku
        JCSystem.abortTransaction();
    }
25 }
```

Wydruk 24. Tworzenie i usuwanie obiektów (`CreateDel.java`)

W przypadku usuwania apletów możliwe jest automatyczne wywołanie metody **uninstall()**, która powinna zawierać:

— zwolnienie zasobów współdzielonych z innymi apletami
— zachowanie danych (np. potrzebnych innym apletom)
— powiadomienie innych apletów o usunięciu

Podczas implementacji tej metody zalecane jest wykorzystanie transakcji oraz zabezpieczenie przed wyborem apletu podczas jego usuwania (poprzez odpowiednie ustawienie wartości zmiennej globalnej).

Współdzielenie obiektów

Technologia *Java Card* przewiduje tworzenie pomiędzy instancjami apletów tzw. „ścian ogniowych”. Pozwalają one, poprzez mechanizm aktywnego kontekstu, uniknąć sytuacji gdy np.: dane wyciekają z apletu lub poprzez niepoprawną metodę implementacji istnieje możliwość przechwycenia danych z aplikacji kartowej. JCVM w danym momencie może obsługiwać dokładnie jeden kontekst. Istnieje możliwość przełączania kontekstów.

Domyślnie poprzez jeden kontekst obsługiwane są instancje apletów będące w tym samym pakiecie. Oznacza to, że mogą wzajemnie korzystać z danych, które przechowują. Ponadto przy

wywoływaniu metod pomiędzy tymi apletami nie następuje zmiana kontekstu. Osobny, uprzywilejowany kontekst posiada JCRE, co pozwala na wykonywanie operacji niedostępnych dla apletów (np. usunięcie pakietu). W przypadku instancji apletów będących w różnych kontekstach podczas wywołania metody pomiędzy apletami (jeżeli aplet udostępnia metodę) nastąpi zmiana kontekstu (wyjątkiem są publiczne metody statyczne klas). Aktualny i poprzedni kontekst wykonania można uzyskać korzystając z metod **getAID()** oraz **getPreviousContextAID()** z klasy **JCSystem**.

Wywołanie metod pomiędzy apletami znajdującymi się w różnych kontekstach jest możliwe po zastosowaniu mechanizmu współdzielenia obiektów. Współdzielone metody powinny być wyszczególnione w interfejsie, który następnie powinien być zaimplementowany. Komunikacja jest oparta na modelu klient-serwer.

```
// przykład interfejsu obiektu
public interface SampleInterface extends Shareable
{
    // metoda, która może być wywoływana przez inny aplet
    void getName(byte[] buffer);
}

// klasa implementująca interfejs SampleInterface
public class SampleClass extends Applet implements SampleInterface
{
    // ...
    // implementacja wymaganych metod klasy (install itp.)
    // ...

    // metoda pozwala na uzyskanie referencji do klasy w celu wywołania
    // metod interfejsu
    public Shareable getShareableInterfaceObject(AID clientAID, byte parameter)
    {
        // jeżeli clientAID oraz parameter spełnia odpowiednie warunki zwracamy referencję
        // do instancji klasy SampleClass
        if (parameter==(byte)0)
            return this;
        // w przeciwnym wypadku zwracamy null
        else return null;
    }

    // metoda z interfejsu
    public void grantPoints(byte[] buffer)
    {
        // ...
        // implementacja metody
        // ...
    }
}

// klasa wykorzystująca obiekt SampleClass
public class SampleClient extends Applet
{
    // obiekt współdzielony
    private SampleInterface shared;

    public void sampleMethod (...)
    {
        // pobranie AID obiektu, którego metody chcemy wywoływać
        AID sampleAID=JCSystem.lookupAID(...)

        // pobranie referencji do apletu
        shared=(SampleInterface)
            JCSystem.getAppletShareableInterfaceObject(sampleAID, (byte)0);

        // sprawdzenie poprawności uzyskanej referencji
        if (shared==null)
        {
```

```
                // nie udalo się uzyskać dostępu do apletu
55          }
            else
            {
                // wywołanie metody apletu
                shared.getName(...);
60          }
        }
    }
```

Wydruk 25. Współdzielenie obiektów (`Sharing.java`)

JCRE udostępnia użytkownikowi, nie posiadającemu specjalnych uprawnień, obiekty, które znajdują się w jego kontekście (*JCRE Entry Point Objects*). Wśród tych obiektów wyróżniamy obiekty tymczasowe (np. obiekt reprezentujący APDU) oraz stałe (np. obiekty reprezentujące AID instancji apletów).

Kanały logiczne

Java Cards obsługują do czterech kanałów logicznych. Pozwalają one na niezależną, współbieżną komunikację z różnymi (lub tym samym apletem) podczas jednej sesji komunikacyjnej. W przypadku różnych apletów nie są potrzebne specjalne zabiegi prowadzące do wykorzystania kanałów logicznych. Należy jedynie w metodzie **process()** przewidzieć możliwość otrzymywania komend z różnych kanałów logicznych. W przypadku apletów, które mogą być kilkakrotnie wybrane w tym samym czasie przy wykorzystaniu wielu kanałów konieczna jest specjalna implementacja.

W przypadku tworzenia apletów zgodnych z ISO 7816-4 należy zwrócić uwagę na odpowiednią implementację zachowania w przypadku wykorzystania komend z serii MANAGE CHANNEL.

```
// klasa obsługująca kanały logiczne
public class LogChannel extends Applet implements MultiSelectable
{
    public boolean select()
5   {
        return true;
    }

    public void deselect()
10  {
        // ...
    }

    // metoda pozwala na wielokrotny wybór apletu; wartość
15  // appInstAlreadySelected jest prawdą, gdy aplet już pracuje na jedym
    // z kanałów
    //
    public boolean select(boolean appInstAlreadySelected)
    {
20      // kanał przez który przyszła aktualna komenda
        byte actualChannel=APDU.getCLAChannel();
        // kanał, na którym aplet został wybrany
        byte selectChannel=JCSystem.getAssignedChannel();
        // ...
25      // jest to komenda SELECT FILE
        if (actualChannel==selectChannel)
            // ...

        if (actualChannel==0)
30      {
            // komenda MANAGE CHANNEL z kanału 0
        }
        else
```

```
            {
35              // zmiana kanału - wymagane jest skopiowanie aktualnego
                // stanu bezpieczeństwa związanego z kanałem
            }
            // ...
        }
40
        // metoda informująca aplet o zakończeniu pracy przez dany kanał
        // logiczny; wartość appInstStillSelected jest prawdą, gdy aplet
        // pracuje jeszcze na innym kanale
        public void deselect(boolean appInstStillSelected)
45      {
            // ...
        }
    }
```

Wydruk 26. Kanały logiczne (`LogChannel.java`)

Charakterystyczne komendy służące do zarządzania kanałami logicznymi to:

— **MANAGE CHANNEL OPEN** – umożliwia otwarcie kanału r korzystając z kanału q

```
0x0q 0x70 0x00 0x0r 0x00 0x00
```

— **MANAGE CHANNEL CLOSE** – zamknięcie kanału r z otwartego kanału q; zamykać wolno wyłącznie kanały 1, 2 oraz 3

```
0x0q 0x70 0x80 0x0r 0x00 0x00
```

Komunikacja w architekturze RMI

Specyfikacja *Java Card* umożliwia tworzenie aplikacji w architekturze RMI (ang. *Remote Method Invocation*). Rozwój aplikacji wygląda podobnie do typowych, rozproszonych aplikacji tworzonych w tej architekturze. Pierwszym krokiem jest utworzenie interfejsu klasy, która udostępnia pewne metody. Należy przy tym pamiętać, że każda ze zdalnych metod może zgłosić wyjątek **RemoteException**. Interfejs powinien być udostępniony do rozwoju aplikacji klienta.

```
// interfejs zawiera metody, które będą mogły być wywoływane poprzez RMI
public interface Loyality extends Remote
{
    // kody błędów
5   public static final short UNDERFLOW=(short)0x6700;
    public static final short OVERFLOW=(short)0x6701;
    public static final short BAD_ARGUMENT=(short)0x6702;

    // maksymalna liczba punktów
10  public static final short MAX_POINTS = (short)2000;

    // odczytanie ilości punktów
    public short read() throws RemoteException;
    // dodanie punktów
15  public void add(short m) throws RemoteException, UserException;
    // odjęcie punktów
    public void sub(short m) throws RemoteException, UserException;
}
```

Wydruk 27. Interfejs Loyality (`Loyality.java`)

Po zdefiniowaniu interfejsu, należy utworzyć klasę, która będzie implementować określone metody. Kolejnym krokiem jest wygenerowanie zaślepki (ang. *stub*), które będą używane przez klienta zdalnego interfejsu. Służy do tego standardowe narzędzie *rmic*, które należy używać koniecznie z opcją *-v1.2*.

```
// implementacja interfejsu Loyality
public class LoyalityImpl extends CardRemoteObject implements Loyality
{
    // przechowuje ilość dostępnych punktów
    private short points=0;

    // konstruktor
    public LoyalityImpl()
    {
        super();
    }

    // dodanie punktów
    public void add(short m) throws RemoteException, UserException
    {
        if (m<=0) UserException.throwIt(BAD_ARGUMENT);
        if ((short)(points+m)>MAX_AMOUNT) UserException.throwIt(OVERFLOW);
        points+=m;
    }

    // odjęcie punktów
    public void sub(short m) throws RemoteException, UserException
    {
        if (m<=0) UserException.throwIt(BAD_ARGUMENT);
        if ((short)(points-m)<0) UserException.throwIt(UNDERFLOW);
        points-=m;
    }

    // odczyt ilości punktów
    public short read() throws RemoteException
    {
        return points;
    }
}
```

Wydruk 28. Implementacja interfejsu Loyality (`LoyalityImpl.java`)

W aplecie kartowym, który będzie obsługiwał żądania klienta konieczne jest utworzenie instancji klasy z zaimplementowanymi metodami oraz utworzenie dyspozytora, który komendy APDU będzie realizował poprzez wywołania zdalnych metod.

```
// klasa reprezentująca aplet
public class LoyalityApplet extends javacard.framework.Applet
{
    // dyspozytor
    private Dispatcher dispatcher;
    // usługa
    private RemoteService service;
    // zdalny interfejs
    private Remote loyality;

    // konstruktor
    public PurseApplet()
    {
        loyality=new LoyalityImpl();

        // stworzenie dyspozytora i powiązanie z nim usługi
        dispatcher=new Dispatcher((short)1);
        service=new RMIService(loyality);
        dispatcher.addService(service, Dispatcher.PROCESS_COMMAND);

        // rejestracja apletu w JCRE
        register();
    }

    // instalacja apletu
    public static void install(byte[] aid, short s, byte b)
```

```
        {
            // stworzenie instancji apletu
            new LoyalityApplet();
        }

        // przetwarzanie komend APDU
        public void process(APDU apdu) throws ISOException
        {
            // przetwarzanie otrzymanej komendy (konwersja i wywołanie metody)
            dispatcher.process(apdu);
        }
    }
```

Wydruk 29. Aplet Loyality (`LoyalityApplet.java`)

Korzystanie z *Java Card RMI* po stronie klienta jest wyjątkowo łatwe i czytelne. Po stworzeniu referencji do obiektu kartowego traktowany jest on jak obiekt lokalny i możliwe jest wywoływanie jego metod. Poniższy przykład został napisany z użyciem biblioteki *Open Card Framework* (zobacz również 6.4).

```
// przykładowy klient
public class OCFLoyalityClient
{
    // AID apletu Loyality
    private static final byte[] LOYALITY_AID = ...

    public static void main(String[] argv) throws RemoteException
    {
        try
        {
            // aktywacja biblioteki OCF
            SmartCard.start();
            // oczekiwanie na kartę
            CardRequest cr = new CardRequest(CardRequest.NEWCARD, null,
                                             OCFCardAccessor.class);
            SmartCard myCard = SmartCard.waitForCard(cr);
            // utworzenie instancji OCFCardAccessor dla Java Card RMI
            CardAccessor myCS = (CardAccessor)
                myCard.getCardService(OCFCardAccessor.class, true);
            // utworzenie instancji Java Card RMI
            JavaCardRMIConnect jcRMI = new JavaCardRMIConnect(myCS);
            // wybranie apletu
            jcRMI.selectApplet(LOYALITY_AID);
            // referencja do obiektu na karcie
            Loyality loyalityApplet=(Loyality)jcRMI.getInitialReference();
            if (loyalityApplet == null)
                throw new Exception("FAILED");

            // wykonanie operacji odczytu ilości punktów z karty
            short points = loyalityApplet.read();
            // dodanie punktów
            loyalityApplet.add((short) 200);
            // odjęcie punktów
            loyalityApplet.sub((short) 20);
            // ponowne odczytanie ilości punktów
            points = loyalityApplet.read();
        }
        catch (UserException e)
        {
            // wyjątek użytkownika (zła operacja na punktach)
            e.printStackTrace();
        }
        catch (Exception e)
        {
            // inne wyjątki
            System.out.println(e);
        }
```

```
            finally
            {
50              try
                {
                    // dezaktywacja biblioteki OCF
                    SmartCard.shutdown();
                }
55              catch (Exception e)
                {
                    System.out.println(e);
                }
            }
60      }
    }
```

Wydruk 30. Klient apletu Loyality - OCF (`OCFLoyalityClient.java`)

W technologii J2ME (*Java 2 Micro Edition*) możliwe jest korzystanie z interfejsu JCRMI w bibliotece SATSA (*Security and Trust Services*, zobacz również 6.5).

```
    // fragment metody korzystającej z SATSA-JCRMI
    try
    {
        // łącze do apletu na karcie w slocie 0 o danym AID
5       String loyalityUrl="jcrmi:0;AID=a0.0.0.1.1.1.1.1.1.1";

        JavaCardRMIConnection jcRMI=(JavaCardRMIConnection)Connector.open(loyalityUrl);

        Loyality loyalityApplet=(Loyality) jcRMI.getInitialReference();
10
        // wykonanie operacji odczytu ilości punktów z karty
        short points = loyalityApplet.read();
        // dodanie punktów
        loyalityApplet.add((short) 200);
15      // odjęcie punktów
        loyalityApplet.sub((short) 20);
        // ponowne odczytanie ilości punktów
        points = loyalityApplet.read();

20      // zamknięcie połączenia
        jcRMI.close();
    }
    catch (UserException e)
    {
25      // wyjątek użytkownika (zła operacja na punktach)
        e.printStackTrace();
    }
    catch (Exception e)
    {
30      // inne wyjątki
        System.out.println(e);
    }
    finally
    {
35      try
        {
            // zamknięcie połączenia
            jcRMI.close();
        }
40      catch (Exception e)
        {
            System.out.println(e);
        }
    }
```

Wydruk 31. Klient apletu Loyality - SATSA (`SATSALoyalityClient.java`)

Zaprezentowany przykład apletu nie jest implementacją bezpieczną. Zdalne metody powinien wywoływać klient, który został uwierzytelniony. Implementując interfejs **SecurityService** można określić szczegółowe zasady dostępu do apletu. Proces jest obsługiwany automatycznie poprzez dyspozytora. Po stronie klienta należy zaimplementować odpowiednio własny obiekt wywodzący się z klasy **OCFCardAccessor**.

7.2.6. Bezpieczeństwo

Możliwość ładowania i wykonywania kodu na karcie (w szczególności dla kart wieloaplikacyjnych w systemach otwartych) niesie ze sobą nowe zagrożenia w porównaniu do kart natywnych w podobnych zastosowaniach. Technologia Java od samego początku istnienia skierowana była na wprowadzenie skutecznych mechanizmów zapewnienia bezpieczeństwa. Podobnie jest z produktem *Java Card*.

Zdefiniowano kilka grup możliwych zagrożeń i związanych z nimi wymagań wobec bezpieczeństwa:

— zrządca karty – minimalne wymagania bezpieczeństwa określające metody dostępu do karty, zarządzanie jej zawartością,
— profil karty (ang. *Smart Card Profile, SCP*) – wymagania wobec natywnego systemu karty oraz JCVM i JCRE, a także warstwy pośredniczącej pomiędzy systemem natywnym a maszyną wirtualną; celem jest niezależność działania apletu od zastosowanego produktu,
— jądro systemu – podstawowe wymagania wobec środowiska systemu *Java Card* takie jak API oraz ściany ogniowe pomiędzy aplikacjami,
— weryfikacja kodu bajtowego – dotyczy poprawności kodu bajtowego jaki ma być umieszczony w karcie,
— bezpieczne ładowanie apletów – weryfikacja statyczna i dynamiczna kodu bajtowego na karcie, problematyka bezpieczeństwa kart nie posiadających możliwości weryfikacji kodu; ładowanie apletów, które nie zostały zweryfikowane poza kartą lub zmienione po weryfikacji,
— instalacja – problemy związane z instalacją (utworzeniem instancji apletu w karcie) tuż przed jego pierwszym wykonaniem,
— usuwanie apletu – aspekty bezpieczeństwa związane z usuwaniem aplikacji z karty,
— usuwanie obiektów – problemy zarządzania pamięcią maszyny wirtualnej Java, bezpieczeństwo usuwania obiektów apletu,
— zdalne wywoływanie metod (ang. *Remote Method Invocation, RMI*) – problemy związane z operacjami na karcie w architekturze RMI, bezpieczeństwo komunikacji,
— kanały logiczne – umożliwiają wykonywanie i komunikację z kilkoma apletami jednocześnie na jednej karcie; problematyka bezpieczeństwa podczas korzystania z kanałów logicznych.

Grupy zagrożeń odnoszą się bezpośrednio do cyklu życia apletu: implementacji, ładowania, użytkowania i usuwania aplikacji. Nie wszystkie wymagania wobec bezpieczeństwa muszą być spełnione w danej implementacji *Java Card*. Wyróżnia się konfiguracje, które określają zestaw wymagań związany z wymienionymi grupami bezpieczeństwa. Zaliczamy do nich:

— konfiguracja minimalna (ang. *Minimal Configuration Protection Profile*) – minimalny zestaw zaimplementowanych grup wymagań wobec bezpieczeństwa; dotyczy jedynie zarządcy karty oraz jądra systemu,
— system *Java Card 2.1.1* (ang. *Java Card System Standard 2.1.1 Configuration Protection Profile*) – zgodna ze specyfikacją *Java Card 2.1.1*; oprócz konfiguracji minimalnej zawiera wymagania wobec instalacji, weryfikacji kodu bajtowego i bezpiecznego ładowania apletów,
— system *Java Card 2.2* (ang. *Java Card System Standard 2.2 Configuration Protection Profile*)

– zgodna ze specyfikacją *Java Card 2.2* i dalszymi (aktualnie *Java Card 2.2.1*); jest to system *Java Card 2.1.1* rozszerzony o zdalne wywoływanie metod, kanały logiczne, usuwanie obiektów i apletów,

— konfiguracja obronna (ang. *Defensive Configuration Protection Profile*) – konfiguracja dla systemów o podwyższonych wymaganiach bezpieczeństwa; jest to system *Java Card 2.2* wraz z bezpiecznym ładowaniem apletów.

7.2.7. Cyberflex

Cyberflex jest kartą firmy *Schlumberger*. Karty te są zgodne ze specyfikacją *Java Card* firmy *Sun* oraz *GlobalPlatform* (zobacz 12.2). Poszczególne wersje karty różnią się ilością dostępnej pamięci oraz interfejsem komunikacyjnym (realizacja tradycyjna oraz w technologii *eGate*).

Rozwój aplikacji dla tych kart może być wspomagany przez specjalne środowisko. Do najważniejszych narzędzi jakie wchodzą w jego skład należy zaliczyć:

— *Schlumberger Smart Card Toolkit* – to główna aplikacja środowiska;
Pozwala na podgląd zasobów (czytniki, karty) zainstalowanych w systemie. Dla kart firmy *Schlumberger* możliwy jest podgląd plików i pełna obsługa operacji na nich z poziomu menu. Dla karty *Cyberflex* możliwa jest prosta edycja pakietów (ładowanie, usuwanie) w karcie oraz instancji apletów (tworzenie, usuwanie).

— *Program File Generator* – aplikacja przeznaczona do automatycznej konwersji plików **.class* (jest to interfejs graficzny dla programu *converter*);

— *APDU Manager* – pozwala na przesyłanie do karty komend APDU.
Aplikacja umożliwia także tworzenie nowych komend oraz edycję i wykonywanie prostych skryptów.

Środowisko umożliwia szczegółowy podgląd danych przesyłanych pomiędzy kartą a aplikacją.

7.2.8. GemXpresso

GemXpresso jest kartą z wirtualną maszyną języka *Java* firmy *Gemplus*. Karta występuje w kilku wersjach o różnej funkcjonalności. Karty są zgodne ze specyfikacją *Java Card* firmy *Sun* oraz *GlobalPlatform* (zobacz 12.2).

Zestaw narzędzi ułatwiający rozwój aplikacji dla tych kart jest rozpowszechniany pod wspólną nazwą *GemXpresso RAD III*. Jest to środowisko stworzone w języku *Java* z użyciem *Open Card Framework* (zobacz 6.4). W jego skład wchodzą:

— *JCardManager* – umożliwia ładowanie plików *CAP* na kartę oraz sprawdzanie jej zawartości i wymianę komend z kartą (lub z symulatorem);
Jest to główna aplikacja pakietu. Z jej poziomu możliwe jest wywołanie pozostałych narzędzi. Wbudowano w nią standardowe komendy obsługiwane przez kartę. Możliwa jest również edycja własnego zestawu komend.

— *GxpConverter* – konwertuje skompilowany aplet na zestaw plików, które mogą być ładowane na kartę;
Jest to aplikacja analogiczna do narzędzia *converter* z pakietu firmy *Sun*. Na podstawie informacji zapisanych w pliku **.gxp* w formacie XML (lista klas, AIDapletu lub pakietu) dokonywana jest konwersja plików **.class*.

— *GxpLoader* – pozwala ładować aplikacje na kartę (lub do jej symulatora);

— *DeploymentEditor* – umożliwia edycję plików zawierających skrypt złożony z komend wykonywanych na karcie;
Funkcjonalność edytora jest mocno rozbudowana. Pliki **.gxd* zawierające skrypty są w formacie XML. Możliwe jest stworzenie własnych klas określających pewne komendy i wykorzystanie ich w skryptach.

— *GSE* – symulator karty;
Symulator jest procesem nasłuchującym na określonym porcie i emulującym czytnik kart procesorowych. W każdej chwili możliwe jest podejrzenie zawartości karty oraz wartości zmiennych klasy. Jest to niezbędna aplikacja podczas testowania i debugowania apletu.
— *CapDump* – wyświetlenie informacji zawartych w danym pliku typu *CAP*.

Przechowywanie konfiguracji i skryptów w plikach o formacie XML pozwala na ich łatwą i intuicyjną modyfikację. Możliwa jest ich edycja bez narzędzi okienkowych. Środowisko jest również w pełni funkcjonalne przy wykorzystaniu jedynie aplikacji konsolowych. Nie ma konieczności uruchamiania interfejsu graficznego.

Uwagi bibliograficzne

Tematyka projektowania aplikacji dla kart natywnych jest szeroko zaprezentowana w [3].

Korzystanie z kart wyposażonych w maszynę wirtualną języka Java (specyfikacja, konfiguracja oprogramowania, projektowanie aplikacji) jest tematem publikacji [96, 97, 98, 94, 92, 89, 93, 95]. Bezpieczeństwo tych rozwiązań opisano w [88, 90, 91].

Kartę *Cyberflex* opisano w [87].

8. Karty w systemach rozliczeniowych

Możliwość bezpiecznego przechowywania danych na karcie inteligentnej została dostrzeżona również przez twórców systemów płatniczych. Karty elektroniczne w systemach bankowych są równie często używane jak w przypadku sektora telekomunikacyjnego. Oprócz tradycyjnego zastosowania w postaci karty płatniczej (analogicznej do karty płatniczej z paskiem magnetycznym) istnieją również takie, które są możliwe do realizacji wyłącznie na karcie inteligentnej. Jest nim elektroniczna portmonetka.

W Polsce karty płatnicze z mikroprocesorem nadal stanowią margines rynku. *Kredyt Bank*, jako pierwszy w Polsce, wydał kartę z mikroprocesorem – jest to karta Proton Prisma umożliwiająca m. in. przeprowadzanie transakcji EMV. Zauważalny jest jednak wzrost popularności tego produktu - coraz więcej klientów wybiera kartę elektroniczną i jest ona w ofercie coraz większej ilości banków.

8.1. Transakcje płatnicze z użyciem karty

Transakcja przeprowadzana z użyciem karty elektronicznej jest analogiczna do płatności z wykorzystaniem karty magnetycznej. Informacje przechowywane na karcie służą do zweryfikowania użytkownika w systemie bankowym. W pierwszej fazie sprawdzony jest numer karty (jego prawidłowość, czy nie jest na liście kart zastrzeżonych), a następnie dokonuje się uwierzytelnienia użytkownika (najczęściej przy użyciu kodu PIN). W przypadku karty inteligentnej kod PIN może być przechowywany na karcie i nie ma wtedy konieczności łączenia się z systemem bankowym.

Możemy wyróżnić trzy podstawowe formy płatności dokonywanych przy użyciu kart z mikroprocesorem:

— natychmiastowa – dokonywana jest przy użyciu karty debetowej; należność za usługę jest pobierana z konta użytkownika w momencie transakcji,
— odroczona – przeprowadzana jest z użyciem karty kredytowej; rozliczenie dokonywane jest po transakcji,
— przed transakcją – służą do tego karty z możliwością wcześniejszego doładowania (elektroniczne portmonetki); podczas transakcji z karty pobiera się odpowiednią ilość gotówki.

Inną spotykaną klasyfikacją jest podział na karty debetowe, kredytowe oraz typu *charge*. Karta typu *charge* rozliczana jest w następujący sposób: co pewien okres bank przesyła nam zestaw transakcji, jakich dokonaliśmy z użyciem karty; na opłacenie „długu” mamy określony czas (po jego przekroczeniu najczęściej naliczane są odsetki karne).

Pośród tworzonych na świecie systemów płatniczych możemy wyróżnić systemy otwarte oraz zamknięte. Pierwsze z nich to zazwyczaj wynik współpracy kilku banków. Umożliwia dokonywanie płatności u sprzedawców posiadających terminale rozliczane przez różne podmioty. Przykładem może być system płatniczy *VISA*. Lokalne elektroniczne portmonetki (karta służąca do opłat za ksero w uczelnianej bibliotece) stanowią przykład systemów zamkniętych.

8.1.1. Elektroniczny pieniądz

Coraz więcej ludzi ceni sobie możliwość transakcji bezgotówkowych. Jaki jednak powinien być elektroniczny pieniądz, który możemy przechowywać na karcie inteligentnej, aby był użyteczny? Najważniejszymi jego cechami powinny być:

— bezpieczeństwo – niedopuszczalne jest wprowadzanie do systemu zduplikowanych pieniędzy oraz transakcji, które nigdy nie zaszły w rzeczywistości,
— przetwarzalność – podstawowa właściwość elektronicznego pieniądza - musi istnieć możliwość całkowitego jego przetwarzania przez maszyny elektroniczne,

— przenośność – nie może być związany wyłącznie z określonym nośnikiem (takim jak karta); może być transferowany przy użyciu np. sieci komputerowej,
— podzielność – musi istnieć możliwość zapłaty bez konieczności wydawania reszty,
— decentralizacja – może istnieć możliwość płatności bez ingerencji systemu bankowego np. przelew gotówki bezpośrednio z karty na kartę,
— możliwość kontroli – operator systemu powinien mieć możliwość jego monitorowania, tak by czuwać nad jego bezpieczeństwem i nie dopuszczać do fałszerstw,
— anonimowość – oznacza, że nie powinno być możliwości skojarzenia transakcji oraz osoby która jej dokonała; uniemożliwia to śledzenie właścicieli elektronicznych pieniędzy (gdzie i jakich dokonują płatności),
— legalność – powinien być to legalny środek płatniczy na terenie danego kraju,
— stabilność – kurs elektronicznego pieniądza nie powinien ulegać dużym wahaniom.

Jak można zauważyć niektóre z tych cech są na pierwszy rzut oka sprzeczne względem siebie. Możliwa jest jednak taka realizacja systemu, aby sprostać jak największej ilości z wymienionych zaleceń.

8.1.2. Podstawy architektury systemu płatniczego

Model systemu płatniczego może być stworzony na wiele sposobów. Najczęściej jednak, ze względów ekonomicznych, oparty jest na istniejącej już części systemu bankowego. Można jednak wyróżnić pewne charakterystyczne komponenty systemu płatniczego do których należą:

— system bankowy – składa się z dwóch części:
  — rozliczeniowej – odpowiada za przepływ gotówki pomiędzy sprzedawcami a klientami oraz pomiędzy różnymi systemami bankowymi,
  — zarządzającej – kontroluje czynności administracyjne takie jak: dystrybucja listy kart zastrzeżonych, nowych kluczy kryptograficznych, uaktualnianie oprogramowania na terminalach;
— sieć – zapewnia połączenie systemu bankowego z terminalami,
— terminale – różne urządzenia umożliwiające korzystanie z kart płatniczych; mogą to być przykładowo:
  — terminale POS (ang. *point of sale*) zlokalizowane u sprzedawców,
  — terminale ATM (ang. *automated teller machine*) zwane popularnie bankomatami;
— karty inteligentne – przeznaczone do przechowywania informacji umożliwiających korzystanie z systemu; sprzedawca będzie posiadał kartę umożliwiającą uruchomienie terminala w systemie, klient kartę umożliwiającą dokonywanie płatności.

Komunikacja pomiędzy terminalami a systemem bankowym odbywa się przy użyciu sieci (Internet, telefonia komórkowa, stacjonarna). Połączenie może być stałe lub zestawiane raz na jakiś czas (np. dla sprzedawców w małych miejscowościach). Podczas połączenia terminal przesyła informacje o transakcjach do systemu bankowego natomiast moduł obsługi terminali po stronie banku uaktualnia parametry i oprogramowanie terminala.

8.2. Elektroniczna portmonetka

Jednym z zastosowań kart elektronicznych w systemach płatniczych są aplikacje elektronicznej portmonetki. Możemy do nich zaliczyć wszelkiego rodzaju karty *prepaid* (zarówno bez jak i z możliwością późniejszego doładowania) oraz klasyczne portmonetki (realizujące funkcjonalność typowego rzeczywistego pieniądza). W niektórych publikacjach rozróżnia się aplikację elektronicznej portmonetki od elektronicznego portfela (zobacz [2]). Pierwsza z nich służy do realizacji małych płatności, druga jest zazwyczaj powiązana z kontem bankowym klienta i ma funkcjonalność karty płatniczej.

8.2.1. Działanie systemu

W profesjonalnym i pełnym systemie elektronicznej portmonetki możemy wyróżnić następujące elementy[15]:

— dostawca systemu (ang. *Scheme Provider*) – organizacja zarządzająca systemem elektronicznej portmonetki i określająca reguły jej używania; jest odpowiedzialna za integrację systemu z poszczególnymi dostawcami oraz innymi systemami płatniczymi,
— urząd certyfikacyjny (ang. *Certification Authority*) – odpowiedzialny za generację certyfikatów i przechowywanie kluczy wykorzystywanych w systemie,
— emitent karty (ang. *Card Issuer*) – wystawca karty płatniczej, najczęściej bank lub instytucja finansowa, organizacja płatnicza,
— emitent pieniędzy (ang. *Funds Issuer*) – organizacja odpowiedzialna za doładowania kart (autoryzacja transakcji oraz dystrybucja pieniędzy),
— właściciel karty (ang. *Cardholder*) – osoba korzystająca z systemu elektronicznej portmonetki,
— centrum autoryzacyjne doładowań (ang. *Load Acquirer*) – centrum pośredniczy pomiędzy akceptantem a emitentem karty (organizacją płatniczą); centrum zapewnia autoryzację, czyli potwierdzenie, że bank lub organizacja posiadacza karty przyjmują na siebie obowiązki wynikające z transakcji doładowania karty,
— akceptant (ang. *Merchant*) – osoba lub instytucja, przyjmująca kartą płatniczą należność za dostarczone towary lub usługi,
— centrum autoryzacyjne akceptantów (ang. *Merchant Acquirer*) – odpowiedzialne za rozliczanie transakcji pomiędzy różnymi dostawcami systemów,
— ośrodek przetwarzania (ang. *Processor*) – odpowiedzialny za przepływ informacji pomiędzy elementami systemu elektronicznej portmonetki.

Wobec systemu elektronicznej portmonetki określone są także szczegółowe wymagania dotyczące bezpieczeństwa. Stosowanymi algorytmami kryptograficznymi są RSA oraz 3DES. Podczas operacji o podwyższonym ryzyku wymagane jest uwierzytelnienie *online* użytkownika. Terminale powinny być wyposażone w moduły SAM, a każda z wystawionych kart zabezpieczona unikatowymi kluczami.

Na karcie będącej elektroniczną portmonetką do przeprowadzenia możliwe są następujące operacje:

— doładowanie (ang. *load*) – wyróżniamy dwa rodzaje doładowania:
  — bezpośrednie (ang. *linked load*) – fundusze nie są przenoszone pomiędzy instytucjami finansowymi (oznacza to, że właściciel karty posiada konto u wydawcy karty i tam dokonuje doładowania),
  — pośrednie (ang. *unlinked load*) – wymagany jest kontakt pomiędzy instytucjami finansowymi oraz uwierzytelnienie użytkownika (poprzez podanie kodu PIN);
— wyładowanie (ang. *unload*) – możliwy jest przelew wyłącznie na konto właściciela elektronicznej portmonetki,
— kupno (ang. *purchase*) – transakcja kupna możliwa jest po wzajemnym uwierzytelnieniu terminala i karty,
— wymiana waluty (ang. *currency exchange*) – zmiana waluty jaką operujemy na elektronicznej portmonetce pod kontrolą wydawcy karty,
— anulowanie ostatniej transakcji kupna (ang. *cancel last purchase transaction*).

[15] na podstawie normy EN 1546, na której bazuje dokument [77]

8.2.2. Mondex

Mondex jest praktycznym przykładem realizacji w pełni działającej elektronicznej portmonetki. System działa w Wielkiej Brytanii. Jego idea powstała w 1990 roku. W praktyce używany jest od 1995 roku.

Użytkownicy posiadają karty na których mogą przechowywać fundusze w maksymalnie pięciu walutach. Wraz z kartami wydawane jest specjalne urządzenie (podobne do kieszonkowego kalkulatora) umożliwiające sprawdzenie wielkości funduszów na karcie oraz transfer środków pomiędzy kartami bez udziału banku. Użytkownik ma możliwość zabezpieczenia wykonywania transakcji kodem PIN.

Karty *Mondex* mogą być używane w terminalach płatniczych oraz doładowywane w publicznych automatach telefonicznych. Użytkownicy wnoszą niewielką miesięczną opłatę za możliwość korzystania z systemu. Opłata taka dotyczy także każdej wykonanej transakcji.

Mondex jest zrealizowany na kartach z systemem operacyjnym *MULTOS* (zobacz 4.10.4).

8.2.3. Realizacja na karcie MPCOS

Karta **MPCOS** jest dedykowaną kartą przeznaczoną dla systemów elektronicznej portmonetki.

Zakładamy, że chcemy stworzyć elektroniczną portmonetkę, która w zabezpieczony sposób porozumiewa się z terminalem. Nie ma możliwości jej użycia poza dedykowanymi terminalami. Karta powinna także przechowywać dane kilku ostatnich transakcji oraz dane użytkownika.

W określonym katalogu należy utworzyć pliki (niektóre z nich są specjalnymi plikami systemu operacyjnego i do operacji na nich przeznaczone są specyficzne komendy):

plik zarządcy transakcji to plik o specjalnej strukturze, którego zadaniem jest przechowywanie stanu licznika transakcji oraz jego kopii zapasowej,

plik z kodami PIN wykorzystywanymi podczas transakcji doładowania i zakupu,

plik elektronicznej portmonetki przeznaczony do przechowywania aktualnego stanu konta oraz jego kopii zapasowej,

pliki z kluczami kryptograficznymi przeznaczonymi do nawiązania zabezpieczonej komunikacji podczas: doładowania, zakupu, uwierzytelnienia użytkownika, odczytu stanu konta oraz administracji kartą,

plik z kluczem kryptograficznym przeznaczonym do podpisywania transakcji,

plik rekordowy do przechowywania ostatnich transakcji,

plik rekordowy do przechowywania informacji o właścicielu elektronicznej portmonetki oraz innych danych.

Podczas personalizacji odpowiednie klucze powinny być tworzone dla danej karty (dywersyfikacja klucza) na podstawie klucza matki.

Dla kart **MPCOS** istnieją odpowiednie moduły SAM (ang. *secure application module*), które pozwalają na bezpieczne przechowywanie kluczy i komunikację w terminalach płatniczych. Mają one możliwość dywersyfikacji kluczy oraz obliczania odpowiednich kryptogramów związanych z komunikacją. Dywersyfikacja polega na utworzeniu klucza kryptograficznego dla danej karty na podstawie klucza matki (zaszytego np. w terminalu płatniczym) i numeru karty (lub innej charakterystycznej informacji).

Dzięki stworzeniu odrębnego klucza dla określonych operacji możliwe jest rozróżnienie terminali, w których użytkownik ma możliwość np. doładowania lub jedynie zakupu czy odczytu stanu portfela. Bez posiadania odpowiedniego klucza wykonanie chronionych operacji będzie niemożliwe.

Wykorzystanie kodu PIN zapewnia bezpieczeństwo środków w przypadku zgubienia lub

kradzieży karty. Konieczność znajomości kluczy administracyjnych wyklucza także nieuprawnione modyfikacje zawartości karty.

Karta ma możliwość przechowywania danego klucza kryptograficznego w czterech wersjach co pozwala np. co określony czas (najczęściej jest to jeden rok) lub w przypadku kompromitacji klucza matki na dalsze korzystanie z karty. Zmiany wiążą się tylko ze stosowanymi modułami SAM.

8.2.4. Aplet elektronicznej portmonetki

Na wydruku 32 zaprezentowano prosty aplet elektronicznej portmonetki. Pozwala on na przechowywanie do 30000 jednostek (np. groszy, co daje kwotę 300 PLN). Operacja doładowania pieniędzy chroniona jest kodem PIN, który powinien znać wyłącznie operator portmonetki. Wydanie pieniędzy nie wymaga znajomości żadnego kodu (wystarczy posiadać kartę). Możliwe jest również sprawdzenie ilości zasobów na karcie. Wartości odpowiednich instrukcji można odczytać z kodu źródłowego. Instrukcje, które pobierają dane (weryfikacja kodu PIN, doładowanie, wydanie pieniędzy) oczekują ich w polu z danymi. Kod PIN ma długość czterech bajtów, natomiast kwoty podawane są w polach dwubajtowych. Parametrem podawanym podczas instalacji jest kod PIN.

Aplet został przetestowany na karcie *Cyberflex* w środowisku *Schlumberger Smart Card Toolkit* (zobacz 7.2.7). W celu wygenerowania odpowiednich plików należy skompilować aplet:

```
javac -target 1.1 -d . -classpath jc_api_212.jar PurseExample.java
```

a następnie dokonać jego konwersji (opcje przygotowane w odpowiednim pliku) oraz stworzyć plik *ijc*, wymagany przy ładowaniu apletu na kartę. Można to zrobić ręcznie:

```
java com.sun.javacard.converter.Converter -config .\converter.opt
java com.slb.javacard.jctools.ijc.MakeIJC -verbose -expFileDir \exp \
    -type onCardVerifier PurseExample.jar
```

lub z użyciem programu *Program File Generator*.

```
// nazwa pakietu
package pl.edu.pw.ii.scproj.samples.PurseExample;

// import niezbędnych pakietów
import javacard.framework.*;

// klasa PurseExample realizująca funkcjonalność elektronicznej portmonetki
public class PurseExample extends Applet
{
    // stałe wykorzystywane przy analizie komend APDU

    // klasa komend APDU elektronicznej portmonetki
    private final static byte CLA_PURSE         = (byte) 0x90;
    // pobranie informacji o zasobach w portfelu
    private final static byte CMD_GET_BALANCE = (byte) 0x50;
    // dodanie zasobów
    private final static byte CMD_CREDIT        = (byte) 0x52;
    // wydanie pieniędzy
    private final static byte CMD_DEBIT         = (byte) 0x54;
    // weryfikacja kodu PIN
    private final static byte CMD_VERIFY_PIN   = (byte) 0x56;

    // kod odpowiedzi po niepoprawnym wprowadzeniu kodu PIN
    private final static short SW_WRONG_PIN = (short) 0x63C0;

    // maksymalna wartość ilości pieniędzy w portmonetce (300,00 PLN)
    private static final short maxBalance = 30000;

    // aktualna ilość pieniędzy w portmonetce
    private short balance;
```

```
    // kod PIN
    private OwnerPIN creditPIN;

35  // stworzenie instancji apletu na karcie
    private PurseExample(byte[] buffer, short offset, byte length)
    {
        short paramOffset = offset;

40      if(length > 9)
        {
            // parametry instalacyjne przekazane zgodnie z OpenPlatform 2.0.1

            // przesunięcie do danych określających prawa apletu
45          paramOffset += (short)(1 + buffer[offset]);
            // przesunięcie do parametrów aplikacji
            paramOffset += (short)(1 + buffer[paramOffset]);
            // oczekiwany jest PIN o długości 4 bajtów
            if(buffer[paramOffset] != 4)
50              // operacja kończy się z błędem
                ISOException.throwIt(
                    (short)(ISO7816.SW_WRONG_LENGTH+offset+length-paramOffset));
            // dane aplikacji (kod PIN)
            paramOffset++;
55      }
        else
        {
            // parametry instalacyjne przekazane zgodnie z OpenPlatform 2.0.
            if(length != 4)
60              ISOException.throwIt((short)(ISO7816.SW_WRONG_LENGTH + length));
        }

        // stworzenie instancji klasy OwnerPIN: kod PIN o długości 4,
        // ilość prób wprowadzenia - 3
65      creditPIN = new OwnerPIN ((byte)3, (byte)4);

        // ustalenie nowego kodu PIN (na podstawie danych podanych przy tworzeniu instancji
        creditPIN.update(buffer, paramOffset,(byte)4);

70      // początkowa ilość pieniędzy w portmonetce
        balance = 0;

        // rejestracja instancji apletu
        register(buffer, (short)(offset+1) ,(byte)(buffer[offset]));
75  }

    // stworzenie instancji apletu
    public static void install(byte[] bArray, short bOffset, byte bLength)
        throws ISOException
80  {
        // wywołanie konstruktora
        new PurseExample (bArray, bOffset, (byte)bLength);
    }

85  // wywoływane po wybraniu apletu
    public boolean select()
    {
        return true;
    }
90
    // wywoływane po zakończeniu korzystania z apletu
    public void deselect()
    {
        // zerowanie statusu kodu PIN
95      creditPIN.reset();
    }

    // przetwarzanie komend APDU
    public void process(APDU apdu) throws ISOException
```

```
100     {
            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();
            // ilość danych w buforze
            byte dataLength = apduBuffer[ISO7816.OFFSET_LC];
105
            // przechwycienie komendy SELECT FILE
            if (selectingApplet())
                ISOException.throwIt(ISO7816.SW_NO_ERROR);

110         // nie obsługujemy kanałów logicznych
            apduBuffer[ISO7816.OFFSET_CLA] =
               (byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);

            // sprawdzenie poprawności klasy instrukcji
115         if (apduBuffer[ISO7816.OFFSET_CLA] != CLA_PURSE)
                ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

            // w zależności od przesłanej instrukcji podejmowane są różne operacje
            switch(apdu.getBuffer()[ISO7816.OFFSET_INS])
120         {
                // pobranie ilości gotówki w portfelu
                case CMD_GET_BALANCE:
                    getBalance(apdu);
                    break;
125
                // doładowanie
                case CMD_CREDIT:
                    creditPurse(apdu);
                    break;
130
                // transakcja
                case CMD_DEBIT:
                    debitPurse(apdu);
                    break;
135
                // weryfikacja kodu PIN
                case CMD_VERIFY_PIN:
                    verifyPIN(apdu);
                    break;
140
                // każda inna instrukcja jest niepoprawna
                default:
                    // zgłoszenie wyjątku
                    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
145                 break;
            }
        }

        // zwraca aktualną ilość gotówki w portfelu
150     private void getBalance(APDU apdu)
        {
            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();

155         // wpisanie aktualnej ilości gotówki (za komendą APDU)
            apduBuffer[5] = (byte)(balance >> 8);
            apduBuffer[6] = (byte)balance;

            // przejście w tryb odpowiedzi
160         apdu.setOutgoing();
            // długość odpowiedzi
            apdu.setOutgoingLength((short)2);

            // przesłanie odpowiedzi APDU (2 bajty od bajtu 5 z bufora)
165         apdu.sendBytes((short)5, (short)2);
        }

        // doładowanie portmonetki
```

```
    private void creditPurse(APDU apdu) throws ISOException
170 {
        // sprawdzenie czy wcześniej podano prawidłowy kod PIN
        if (!creditPIN.isValidated())
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

175     // pobranie bufora z APDU
        byte[] apduBuffer = apdu.getBuffer();

        // sprawdzenie czy użytkownik dostarczy 2 bajty danych (kwota wypłaty)
        // i czy udało się je odebrać
180     if( apduBuffer[4] != 2 || apdu.setIncomingAndReceive() != 2 )
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        // odczytanie wartości doładowania (bajty 5 i 6 w przesłaniej komendzie APDU)
        short amount = (short)(((apduBuffer[5] << 8 ) & (short)0xFF00) +
185                    ((apduBuffer[6]) & (short)0x00FF));

        // spawdzenie poprawności wartości doładowania
        if ((amount > (short)(maxBalance - balance)) || (amount <= (short)0))
            ISOException.throwIt(ISO7816.SW_DATA_INVALID);
190     else
            // doładowanie eklektronicznej gotówki
            balance += amount;
    }

195 // wypłata gotówki
    private void debitPurse(APDU apdu) throws ISOException
    {
        // pobranie bufora z APDU
        byte[] apduBuffer = apdu.getBuffer();
200
        // sprawdzenie czy użytkownik dostarczy 2 bajty danych (kwota wypłaty)
        // i czy udało się je odebrać
        if(apduBuffer[4] != 2 || apdu.setIncomingAndReceive() != 2)
        {
205         // operacja zakończona z błędem
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;
        }

        // odczytanie wartości transakcji (bajty 5 i 6 w przesłaniej komendzie APDU)
210     short amount = (short)(((apduBuffer[5] << 8 ) & (short)0xFF00) +
                       ((apduBuffer[6]) & (short)0x00FF));

        // sprawdzenie czy kwota jest poprawna
        if ((balance >= amount) && (amount > 0))
215     {
            // uaktualnienie stanu portfela
            balance -= amount ;

            // nowy stan portfela zostanie przesłany w odpowiedzi
220         apduBuffer[9] = (byte)(balance >> 8);
            apduBuffer[10] = (byte)balance;

            // przesłanie 2 bajtów odpowiedzi
            apdu.setOutgoing();
225         apdu.setOutgoingLength((short)2);
            apdu.sendBytes((short)9, (short)2);
        }
        else
            // kwota nie jest poprawna
230         throw new ISOException (ISO7816.SW_DATA_INVALID) ;
    }

    // weryfikacja kodu PIN
    private void verifyPIN(APDU apdu)
235 {
        // pobranie bufora z APDU
        byte[] buffer = apdu.getBuffer();
```

```
            // długość kodu PIN
240         byte pinLength = buffer[ISO7816.OFFSET_LC];

            // pobranie przekazanych danych
            if(apdu.setIncomingAndReceive()<pinLength)
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
245
            if(!creditPIN.check(buffer, ISO7816.OFFSET_CDATA, pinLength))
            {
                // sprawdzenie kodu PIN nie powiodło się
                byte triesRemaining = creditPIN.getTriesRemaining();
250             // ilość pozostałych prób wynosi x (63 Cx)
                ISOException.throwIt((short)(SW_WRONG_PIN + triesRemaining));
            }
        }
    }
```

Wydruk 32. Aplet elektronicznej portmonetki (`PurseExample.java`)

8.3. System płatniczy EMV

Ze względu na szybki rozwój płatności elektronicznych koniecznością jest ich standaryzacja zarówno pod względem wymagań wobec bezpieczeństwa transakcji (eliminacja oszustw i nadużyć) jak i budowy oraz funkcjonalności stosowanej karty. Przykładem standardu w tym obszarze wykorzystania kart jest norma EMV(*Europay, MasterCard, Visa*). Jej pełny tytuł brzmi *Integrated Circuit Card Application Specification for Payment System*. Dotyczy ona w szczególności:

— przebiegu transakcji oraz wymagań wobec karty elektronicznej,
— danych przetwarzanych podczas transakcji i przechowywanych na karcie,
— wymagań wobec bezpieczeństwa.

Do aplikacji na karcie zgodnej z EMV możemy odwoływać się poprzez SFI lub nazwę DF (jest ona w tym wypadku tożsama z AID (ang. *application identifier*) aplikacji). Norma przewiduje w katalogu głównym karty plik „1PAY.SYS.DDF01” zawierający informacje o aplikacjach umieszczonych na karcie.

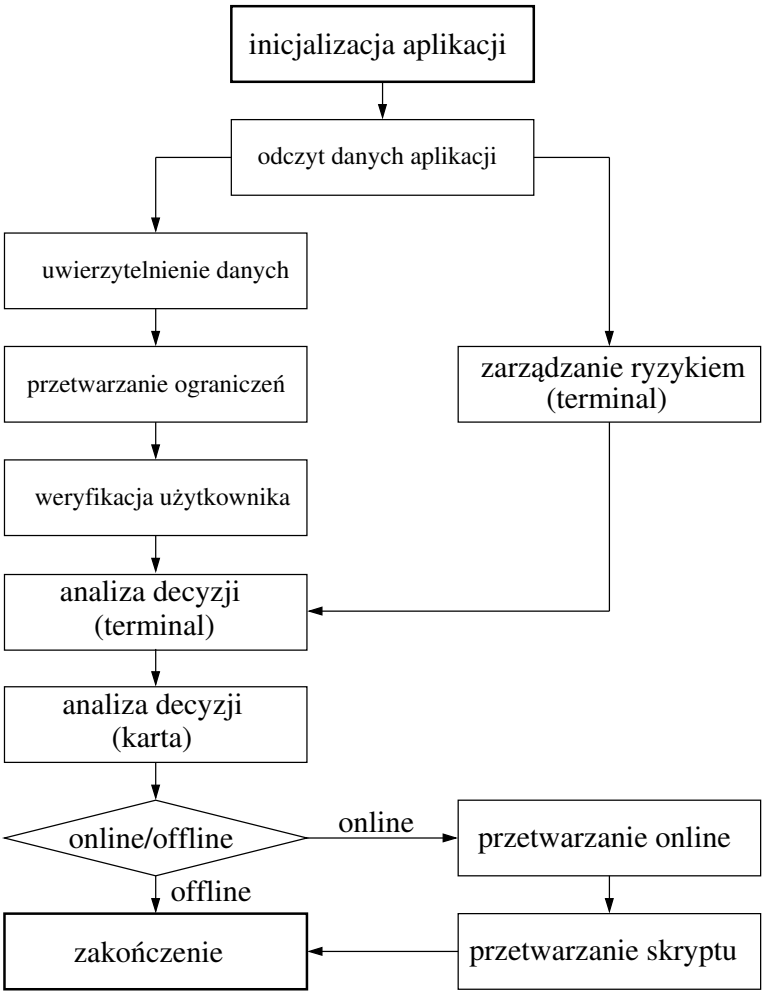
Na rysunku 25 pokazano przebieg transakcji EMV. Składa się on z następujących kroków:

inicjalizacja aplikacji (ang. *Initiate Application Processing*) – karta otrzymuje informacje od terminala dotyczące nowej transakcji; w odpowiedzi zwracane są informacje o plikach na karcie, które mogą być wykorzystane podczas dalszego przebiegu transakcji,

odczyt danych aplikacji (ang. *Read Application Data*) – z karty odczytywane są dane potrzebne do obsłużenia transakcji,

uwierzytelnienie danych (w trybie offline) (ang. *Offline Data Authentication*) – proces uwierzytelnienia przeprowadzany w trybie SDA albo DDA:
— SDA (ang. *Static Data Authentication*) – służy uwierztelnieniu statycznych danych zapisanych na karcie przez jej wydawcę,
— DDA (ang. *Dynamic Data Authentication*) – oprócz danych, które uwierzytelniane są w SDA dodatkowo tej operacji poddaje się kartę oraz dane otrzymane z terminala.
Rezultaty zapisywane są w obiektach TVR (ang. *Terminal Verification Result*) oraz TSI (ang. *Transaction Status Information*); informacje te wykorzystywane są w dalszym przebiegu transakcji (akceptacja, odrzucenie),

przetwarzanie ograniczeń (ang. *Processing Restrictions*) – w fazie tej określana jest zgodność aplikacji terminalowej i kartowej; pod uwagę brane są także inne ograniczenia takie jak: data ważności karty, warunki jej użycia; w przypadku wykrycia niezgodności transakcja nie musi zostać przerwana - istnieje możliwość jej realizacji,



Rysunek 25. Przebieg transakcji w systemie EMV

weryfikacja użytkownika (ang. *Cardholder Verification*) – polega na uwierzytelnieniu użytkownika wybranej wcześniej aplikacji z karty; najczęściej realizowane poprzez podanie kodu PIN (weryfikacja *online* albo *offline*) lub złożenie podpisu,

zarządzanie ryzykiem po stronie terminala (ang. *Terminal Risk Managment*) – w celu eliminacji oszustw terminal decyduje o autoryzacji transakcji (zazwyczaj o dużej wartości lub przez losowy wybór) w trybie *online*,

analiza decyzji po stronie terminala (ang. *Terminal Action Analysis*) – w zależności od decyzji podjętej w poprzednim kroku terminal decyduje o kontynuacji transakcji w trybie *online* lub też o akceptacji bądź odrzuceniu w trybie *offline*,

analiza decyzji po stronie karty (ang. *Card Action Analysis*) – aplikacja na karcie (w zależności od wcześniej ustalonych przez jej wydawcę warunków) akceptuje lub odrzuca transakcję,

przetwarzanie transakcji (w trybie online) (ang. *Online Processing*) – transakcja przetwarzana jest przy udziale zewnętrznego systemu autoryzacji,

przetwarzanie skryptu (ang. *Script Processing*) – etap ten nie jest obowiązkowy; jest to ciąg poleceń dla karty niekoniecznie związanych z aktualną transakcją; wykonany skrypt może mieć wpływ na dalsze funkcjonowanie danej aplikacji na karcie,

zakończenie transakcji (ang. *Completion*) – transakcja została zrealizowana.

Karta na której chcemy zrealizować aplikację EMV musi mieć możliwość realizacji następujących komend:

— APPLICATION BLOCK – blokuje aplikację na karcie,
— APPLICATION UNBLOCK – odblokowuje aplikację na karcie,
— CARD BLOCK – blokuje wszystkie aplikacje na karcie,
— EXTERNAL AUTHENTICATE – uwierzytelnienie terminala przez kartę,
— GENERATE APPLICATION CRYPTOGRAM – stworzenie kryptogramu,
— GET CHALLENGE – generacja liczby pseudolosowej,
— GET DATA – wykorzystywana do odczytu porcji danych,
— GET PROCESSING OPTIONS – inicjalizacja transakcji,
— INTERNAL AUTHENTICATE – uwierzytelnienie karty przez terminal,
— PIN CHANGE/UNBLOCK – służy do zmiany lub odblokowania kodu PIN,
— READ RECORD – odczyt rekordu,
— SELECT FILE – wybór pliku,
— VERIFY – uwierzytelnienie użytkownika (np. sprawdzenie kodu PIN).

Niektóre z tych komend są standardowymi komendami, których specyfikację można znaleźć w ISO/IEC 7816-4. Część jednak spełnia funkcjonalność opisaną w normie, jednak jest zaimplementowana w inny sposób. Jeśli karta posiada odpowiednio zaimplementowane komendy, zgodne ze standardem EMV, mówi się, że jest wyposażona w filtr EMV. Często jest to dodatkowa funkcjonalność np. kart lojalnościowych. Przykładowy szkielet implementacji transakcji EMV z użyciem jądra EMV jest przedstawiony na wydruku 33.

```
// załadowanie do odpowiednich struktur danych aplikacji obsługiwanych przez terminal

// wybór wspólnych aplikacji
switch (EMVApplicationSelect(dane terminala))
{
    case OK:
        // istnieją wspólne aplikacje
        break;
    case FALLBACK :
        // transakcję można kontynuować, ale nie z użyciem karty EMV
        return FALLBACK;
    default :
        // błąd - brak wspólnych aplikacji
        return ERROR;
}

// rozpoczęcie przetwarzania
if (EMVInitiateApplicationProcessing(dane ówsplnych aplikacji)!=OK)
    // przerwanie transakcji
    return ERROR;

// odczytanie danych aplikacji oraz uwierzytelnienie danych
if ((EMVReadApplicationData()!=OK) ||
    (EMVAuthOfflineData()==TERMINATE_TRANSACTION))
    // przerwanie transakcji
    return ERROR;

// przetwarzanie ograniczeń
EMVProcessRestrictions();

// weryfikacja użytkownika
if (EMVVerifyCardHolder()==TERMINATE_TRNANSACTION)
    // przetwarzanie transakcji
    return ERROR;

// zarządzanie ryzykiem po stronie terminala
EMVManageTerminalRisk();

// decyzja po stronie karty i terminala
```

```
40  switch(EMVAnalyzeTerminalAndCardAction())
    {
        case ONLINE:
            // transakcja ma być wykonana w trybie online
            onlineTransaction=TRUE;
45          break;
        case APPROVED:
            // transakcja zatwierdzona
            onlineTransaction=FALSE;
            break;
50      case DECLINED:
            // transakcja została odrzucona
            onlineTransaction=FALSE;
            break;
        case FALLBACK:
55          // transakcję można kontynuować, ale nie z użyciem karty EMV
            return FALLBACK;
        default:
            // przerwanie transakcji
            return ERROR;
60  }

    // transakcja online
    if (onlineTransaction)
    {
65      // przetwarzanie transakcji (online)
        switch(EMVProcessOnlineData())
        {
            case APPROVED:
                // transakcja zatwierdzona
70              break;
            case DECLINED:
                // transakcja została odrzucona
                break;
            case FALLBACK:
75              // transakcję można kontynuować, ale nie z użyciem karty EMV
                return FALLBACK;
            default:
                // przerwanie transakcji
                return ERROR;
80      }
    }

    // zakończenie transkacji - wydruk potwierdzenia dla klienta
```

Wydruk 33. Szkielet implementacji transakcji EMV (`emvtrans.c`)

Algorytmy kryptograficzne stosowane w systemie EMV to 3DES oraz RSA. Przewiduje się także możliwość wykorzystania w przyszłości algorytmów opartych na krzywych eliptycznych.

Implementacja EMV na terminalu płatniczym wiąże się zazwyczaj z wykorzystaniem przygotowanych już funkcji realizujących poszczególne kroki algorytmu płatniczego. Terminal powinien posiadać certyfikat wydawany przez organizację EMVCo na dwóch poziomach: sprzętowym oraz aplikacyjnym.

Zgodność terminala oraz karty płatniczej z systemem EMV daje nam pewność współpracy tych dwóch komponentów, nawet jeśli pochodzą od różnych producentów lub wydawców.

Uwagi bibliograficzne

Kartom wykorzystywanym w bankowości poświęcone są serwisy [105, 100].

Pojecię elektronicznego pieniądza i portmonetki elektronicznej rozwinięto w [3] i [77].

System EMV jest szczegółowo opisany w [33, 34, 35, 36].

9. Aplikacje lojalnościowe

System lojalnościowy polega na związaniu klienta z pewnym producentem lub sprzedawcą poprzez możliwość uzyskania pewnych profitów za kupowanie określonych produktów lub zakupy w danych miejscach. Najczęściej realizowany jest poprzez gromadzenie punktów, które następnie można zamienić na upusty przy zakupach lub darmowe produkty. Jednym ze sposobów realizacji programu lojalnościowego jest wyposażenie każdego z klientów w kartę, na której przechowywane są punkty. W zależności od wymagań karta może być spersonalizowana lub anonimowa.

Przyjmijmy, że mamy do zaprojektowania kartę przeznaczoną dla systemu lojalnościowego w którym udział bierze dwóch partnerów. Punkty sumowane są dla każdego z partnerów oddzielnie, czyli zdobywając punkty u pierwszego z nich nie możemy ich użyć u drugiego. Dodatkowo karta ma pamiętać całkowitą ilość punktów zgromadzonych przez klienta u każdego z partnerów od początku trwania programu. Operacje na karcie mogą być dokonywane wyłącznie w uwierzytelnionych terminalach. Karta także powinna być uwierzytelniana przez terminal. Karty są spersonalizowane, czyli przechowujemy na nich podstawowe informacje o właścicielu. Ponadto karta zawiera dane o 10 ostatnich transakcjach zarówno doładowania jak i wykorzystania punktów.

System zrealizujemy na karcie *GemClub-Micro* oraz karcie z maszyną wirtualną Java (kardlet). Przedmiotem projektu będzie sama karta oraz ogólna idea działania systemu. Nie poruszamy problemów związanych z rozliczaniem transakcji oraz implementacją programu terminala.

9.1. Realizacja na karcie GemClub-Micro

Jak już wspomniano wcześniej (zobacz rozdział 4.10.9) karta *GemClub-Micro* wyposażona jest w system dedykowany aplikacjom lojalnościowym. Posiada zaimplementowane specjalne komendy umożliwiające wykonywanie operacji na karcie.

Struktura plików na karcie dostarczonej od producenta wygląda następująco:

— *System File* – plik zawiera zdefiniowane warunki dostępu do karty jako całości:
  — tworzenie, uaktualnianie i kasowanie plików wymaga znajomości klucza *System Key*,
  — odczyt z karty nie jest zabezpieczony;
— *System Key* – systemowy klucz wprowadzony na kartę przez producenta, służy do zabezpieczenia karty przed nieuprawnioną operacją jej personalizacji (np. w przypadku kradzieży kart z transportu); nie ma możliwości bezpośredniego odczytania klucza z karty, można go jednak zmienić w trybie bezpiecznej wymiany danych wymagającej znajomości klucza *System Key*.

Ponieważ chcemy, aby punkty były również zliczane wspólnie karta powinna zawierać plik wspólnego licznika. Dodatkowo tworzymy dwa pliki liczników dla każdego z partnerów. Z kolei tym plikom odpowiadać powinny pliki reguł zawierające zasady naliczania punktów.

Do przechowywania transakcji posłużą nam dwa pliki z 5 rekordami (pliki cykliczne). Dane o użytkowniku będziemy przechowywać w pliku rekordowym o zmiennej długości rekordu. Poszczególne pola będą zawierały takie informacje jak imię, nazwisko, adres, data urodzenia, data wydania karty oraz data ważności karty.

Operacje tworzenia plików na karcie i operowania na nich nie powinny być dostępne dla wszystkich. Gdyby tak było użytkownicy mogliby np. dodawać sobie samodzielnie punkty. W celu uczynienia naszej karty bezpieczną należy utworzyć odpowiedni system kluczy. Dwa z nich dostarczone są przez producenta. *System Key* chroni plik *System File* oraz plik wspólnego licznika przed operacjami kasowania i nadpisania. *Kevol* przeznaczony jest do ochrony

karty przed nieuprawnionym zakładaniem plików. Dodatkowo utworzymy pliki z kluczami chroniącymi pliki przed usunięciem, modyfikacjami oraz operacjami zarezerwowanymi jedynie dla partnerów programu. Musimy zadbać o to aby przykładowo: drugi partner nie mógł naliczać punktów na rzecz pierwszego czy też użytkownik naliczał sobie punkty poza systemem.

Mamy już ogólną koncepcję działania karty lojalnościowej. Teraz przystąpimy do szczegółowego określenia plików oraz zasad personalizacji i operacji z użyciem karty.

Na karcie umieścimy następujące obiekty:

plik rekordowy z danymi użytkownika w kolejnych rekordach przechowujemy: imię, nazwisko, datę urodzenia, adres oraz datę ważności karty; aby wykonać operację aktualizacji lub usunięcia pliku należy wykazać się znajomością klucza chroniącego pliki z danymi, operacja odczytu nie jest zabezpieczona,

plik z informacjami o ostatnich pięciu transakcjach gromadzenia punktów jest to plik rekordowy cykliczny w którym przechowujemy pięć zestawów rekordów zawierających datę przeprowadzenia transakcji, jej kwotę, liczbę przyznanych punktów oraz podpis transakcji; aby wykonać operację aktualizacji lub usunięcia pliku należy wykazać się znajomością klucza chroniącego pliki z danymi, operacja odczytu nie jest zabezpieczona,

plik z informacjami o ostatnich pięciu transakcjach wymiany punktów plik rekordowy cykliczny, w którym przechowujemy pięć zestawów rekordów zawierających datę operacji oraz liczbę wymienionych punktów; aby wykonać operację aktualizacji lub usunięcia pliku należy wykazać się znajomością klucza chroniącego pliki z danymi, operacja odczytu nie jest zabezpieczona,

wspólny licznik kumulacyjny dla partnerów programu plik ten jest chroniony przed operacjami modyfikacji i usunięcia kluczem *System Key*; operacja bezpośredniego dodania punktów jest zablokowana (są one naliczane przez mechanizm lojalnościowy dla partnerów); wykorzystanie punktów jest możliwe tylko przy znajomości przez terminal klucza związanego z konsumpcją oraz kodu PIN (użytkownik); podpisy transakcji generowane są przy użyciu klucza do podpisywania transakcji; z plikiem skojarzone są reguły naliczania punktów obu partnerów,

licznik punktów pierwszego partnera licznik ten nie jest zabezpieczony przed odczytem, natomiast do operacji uaktualnienia, usunięcia i operacji na liczniku wymagana jest znajomość klucza pierwszego partnera; plik jest związany z regułą pierwszego partnera,

licznik punktów drugiego partnera zasady dostępu i reguły stosowanej do tego pliku są analogiczne jak i pierwszego partnera; drugi partner posiada jednak swoją regułę oraz klucz,

reguła przyznawania punktów u pierwszego partnera plik zawiera reguły naliczania punktów dla wspólnego licznika oraz licznika pierwszego partnera; są to makra w postaci $punkty = P_1 * \lfloor \frac{kwota\,transakcji}{P_2} \rfloor$; specyfikujemy parametry P_1, który oznacza ilość przyznanych punktów za kwotę określoną jako parametr P_2; odczyt reguły nie jest zabezpieczony; wszelkie modyfikacje możliwe są przy znajomości klucza przeznaczonego specjalnie dla tego pliku; by wykorzystać regułę należy znać klucz, który jest przeznaczony do operacji na plikach licznika pierwszego partnera,

reguła przyznawania punktów u drugiego partnera jest to plik o przeznaczeniu analogicznym do poprzedniego; reguły i klucze stosowane są w wersji dla drugiego partnera,

pliki z kluczami klucze przed modyfikacją i usunięciem chronione są samymi sobą, nie ma możliwości odczytania klucza; maksymalna ilość błędnych prezentacji klucza wynosi trzy; na podstawie wcześniejszych informacji na karcie zawarte będą następujące klucze:

— chroniący operacje na wspólnym liczniku punktów,

— chroniący operacje na plikach z danymi (użytkownika, transakcji),

— do generowania podpisów transakcji,

— chroniące plik zasad oraz licznik pierwszego partnera,
— chroniące plik zasad oraz licznik drugiego partnera,
— plik przechowujący kod PIN użytkownika, używany przy wymianie punktów;

W terminalach przechowujemy klucze matki potrzebne do operacji na karcie. Każda z kart posiada klucze wyprowadzone z kluczy matek oraz numeru seryjnego karty. Algorytmem kryptograficznym stosowanym na kartach *GemClub-Micro* jest 3DES. Terminale każdego z partnerów posiadają klucze charakterystyczne tylko dla danego z uczestników programu.

Wydawanie (personalizacja) karty odbywa się u operatora systemu lojalnościowego, który współpracuje z partnerami programu i nim zarządza. Posiada on specjalne klucze umożliwiające zakładanie systemu plików na karcie. Jego zadaniem jest także bezpieczne umieszczenie kluczy w terminalach. Dystrybucję kluczy można przeprowadzić na dwa sposoby:
— jeśli posiadamy terminale, w których klucze mogą być bezpiecznie przechowywane w specjalnym obszarze pamięci, to mogą one tam być umieszczone; wymaga to zazwyczaj dostarczenia terminali do zarządcy systemu, który umieści na nich klucze w bezpiecznym środowisku,
— klucze można umieścić na kartach SAM (ang. *secure application module*) – rozwiązanie to pozwala rozesłać klucze na kartach, które sprzedawcy umieszczają w swoich terminalach; jedynie, gdy posiadamy tą kartę możemy dokonywać transakcji z użyciem kart klientów; dodatkowo moduł SAM odpowiedzialny jest za weryfikację i podpisywanie transakcji.

W przypadku zmiany reguł naliczania punktów konieczna jest ich modyfikacja na wszystkich kartach. Oczywiście można aktualizować oprogramowanie w terminalach, by automatycznie aktualizowało kartowe makra, ale zdaje się to być niepotrzebnym kłopotem. Punkty mogą być obliczane po stronie terminala, a na karcie można umieścić „sztuczną” regułę z parametrami $P_1 = 1$ oraz $P_2 = 1$ dzięki czemu można do karty przesłać bezpośrednio ilość punktów.

Zaprojektowana karta pozwala na odczytanie z niej ilości dostępnych punktów oraz danych o transakcjach i użytkowniku bez konieczności posiadania specjalnego terminala (dostęp do odczytu nie wymaga znajomości żadnego z kluczy). Pozwala to na zaprojektowanie aplikacji na komputer domowy umożliwiającej sprawdzenie danych na karcie jeśli tylko użytkownik posiada czytnik kart inteligentnych.

9.2. Realizacja na Java Card

W zaprezentowanej na wydruku 34 realizacji przyjęto kilka uproszczeń w stosunku do opisu z początku rozdziału: nie zrealizowano dziennika transkacji, partnerzy obsługiwani są tym samym kluczem, wykorzystano algorytm DES. Pozwoliło to uczynić kod źródłowy bardziej czytelnym.

Aplet obsługuje następujące instrukcje w klasie 0x90:
— pobranie aktualnej ilości punktów (0x60) – parametr P1 określa numer partnera (1 lub 2); podanie wartości 3 powoduje zwrot sumarycznej liczby punktów,
— dodanie punktów (0x62) – parametr P1 określa numer partnera (1 lub 2); wymagane jest wcześniejsze uwierzytelnienie terminala,
— odjęcie punktów (0x64) – parametr P1 określa numer partnera (1 lub 2); wymagane jest wcześniejsze uwierzytelnienie terminala oraz użytkownika (poprzez podanie kodu PIN),
— weryfikacja kodu PIN (0x66) – kod PIN o długości 4 przekazywany jest w polu danych,
— generacja liczby losowej (0x68) – liczba losowa o długości 8 bajtów zwracana jest wraz z kodem odpowiedzi; na jej podstawie może być przeprowadzone uwierzytelnienie terminala,
— uwierzytelnienie karty (0x6A) – w polu danych komendy przekazywane jest 8 bajtów danych, które karta zaszyfruje kluczem DES i zwróci w postaci 8 bajtów odpowiedzi,

— uwierzytelnienie terminala (0x6C) – terminal przesyła zaszyfrowaną otrzymaną wcześniej liczbę losową; zwrócone bajty statusu określają czy komenda została wykonana poprawnie.

Podczas instalacji instancji apletu w karcie przekazywany jest kod PIN (4 bajty) oraz klucz służący do uwierzytelnienia (8 bajtów). Brak tych parametrów uniemożliwia stworzenie instancji.

Aplet został przetestowany na karcie *GemXpressoPro R3*.

```
// nazwa pakietu
package pl.edu.pw.ii.scproj.samples;

// import niezbędnych pakietów
import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

// klasa LoyaltyExample realizująca funkcjonalność programu lojalnościowego
public class LoyaltyExample extends Applet
{
    // stałe wykorzystywane przy analizie komend APDU

    // klasa komend APDU apletu lojalnościowego
    private final static byte CLA_LOYALTY      = (byte) 0x90;
    // pobranie informacji o ilości punktów
    private final static byte CMD_GET_POINTS   = (byte) 0x60;
    // dodanie punktów
    private final static byte CMD_ADD_POINTS   = (byte) 0x62;
    // odjęcie punktów
    private final static byte CMD_SUB_POINTS   = (byte) 0x64;
    // weryfikacja kodu PIN
    private final static byte CMD_VERIFY_PIN   = (byte) 0x66;
    // weryfikacja kodu PIN
    private final static byte CMD_ASK_RANDOM   = (byte) 0x68;
    // uwierzytelnienie karty
    private final static byte CMD_INT_AUTH     = (byte) 0x6A;
    // uwierzytelnienie terminala
    private final static byte CMD_EXT_AUTH     = (byte) 0x6C;

    // kod odpowiedzi po niepoprawnym wprowadzeniu kodu PIN
    private final static short SW_WRONG_PIN = (short) 0x63C0;

    // maksymalna wartość ilości punktów
    private static final short maxPoints = 15000;

    // aktualne ilości punktów u partnerów
    private short [] points;

    // ilości punktów u partnerów zgromadzone od początku programu
    private short allPoints;

    // kod PIN
    private OwnerPIN ownerPIN;

    // klucz (obustronne uwierzytelnienie)
    private Key desKey;

    // szyfrowanie algorytmem DES
    private Cipher cipherDES;

    // generator liczb losowych
    private RandomData random;

    // flaga określająca prawa dostępu
    private boolean externalAuth;

    // flaga określająca czy wygenerowano liczbę losową
    private boolean randomGen;

```

```
    // bufor zawierający ostatnio wygenerowaną liczbę losową
    private byte[] lastRandom;

    // stworzenie instancji apletu na karcie
    private LoyaltyExample(byte[] buffer, short offset, byte length)
    {
        short paramOffset = offset;

        desKey = null;
        cipherDES = null;

        if(length > 9)
        {
            // parametry instalacyjne przekazane zgodnie z OpenPlatform 2.0.1

            // przesunięcie do danych określających prawa apletu
            paramOffset += (short)(1 + buffer[offset]);
            // przesunięcie do parametrów aplikacji
            paramOffset += (short)(1 + buffer[paramOffset]);

            // oczekiwany jest PIN o długości 4 bajtów oraz klucz DES o długości 8 bajtów
            if(buffer[paramOffset] != 12)
                // operacja kończy się z błędem
                ISOException.throwIt(
                (short)(ISO7816.SW_WRONG_LENGTH+offset+length-paramOffset));

            // dane aplikacji (kod PIN, klucz)
            paramOffset++;
        }
        else
        {
            // parametry instalacyjne przekazane zgodnie z OpenPlatform 2.0.
            // oczekiwany jest PIN o długości 4 bajtów oraz klucz DES o długości 8 bajtów
            if(length != 12)
                ISOException.throwIt((short)(ISO7816.SW_WRONG_LENGTH + length));
        }

        // stworzenie instancji klasy OwnerPIN: kod PIN o długości 4, ilość prób wprowadzenia - 3
        ownerPIN = new OwnerPIN ((byte)3, (byte)4);

        // ustalenie nowego kodu PIN (na podstawie danych podanych przy tworzeniu instancji)
        ownerPIN.update(buffer, paramOffset,(byte)4);

        // inicjalizacja klucza DES
        paramOffset+=4;
        desKey = KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
                                     KeyBuilder.LENGTH_DES, false);
        ((DESKey)desKey).setKey(buffer, paramOffset);

        // początkowa ilość punktów
        allPoints = 0;
        points = new short[2];
        points[0]=points[1]=0;

        // bufor na liczbę losową
        lastRandom = new byte[8];

        // generator liczb losowych
        random = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);

        // instancja klasy szyfrującej algorytmem DES
        cipherDES = Cipher.getInstance(Cipher.ALG_DES_ECB_NOPAD, false);

        // ustalenie wartości początkowej flag
        resetSecurity();

        // rejestracja instancji apletu
        register(buffer, (short)(offset+1) ,(byte)(buffer[offset]));
    }
```

```
130
        // stworzenie instancji apletu
        public static void install(byte[] bArray, short bOffset, byte bLength)
            throws ISOException
        {
135         // wywołanie konstruktora
            new LoyaltyExample (bArray, bOffset, (byte)bLength);
        }

        // wywoływane po wybraniu apletu
140     public boolean select()
        {
            // ustalenie wartości początkowej flag
            resetSecurity();
            return true;
145     }

        // wywoływane po zakończeniu korzystania z apletu
        public void deselect()
        {
150         // ustalenie wartości początkowej flag
            resetSecurity();
        }

        private void resetSecurity()
155     {
            // zerowanie statusu kodu PIN
            ownerPIN.reset();
            // nie było prawidłowego uwierzytelnienia terminala
            externalAuth=false;
160         // nie wygenerowano liczby losowej
            randomGen=false;
        }

        // przetwarzanie komend APDU
165     public void process(APDU apdu) throws ISOException
        {
            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();
            // ilość danych w buforze
170         byte dataLength = apduBuffer[ISO7816.OFFSET_LC];

            // przechwycenie komendy SELECT FILE
            if (selectingApplet())
                ISOException.throwIt(ISO7816.SW_NO_ERROR);
175
            // nie obsługujemy kanałów logicznych
            apduBuffer[ISO7816.OFFSET_CLA] =
               (byte)(apduBuffer[ISO7816.OFFSET_CLA] & (byte)0xFC);

180         // sprawdzenie poprawności klasy instrukcji
            if (apduBuffer[ISO7816.OFFSET_CLA] != CLA_LOYALTY)
                ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

            // w zależności od przesłanej instrukcji podejmowane są różne operacje
185         switch(apdu.getBuffer()[ISO7816.OFFSET_INS])
            {
                // pobranie ilości gotówki w portfelu
                case CMD_GET_POINTS:
                    getPoints(apdu);
190                 break;

                // doładowanie
                case CMD_ADD_POINTS:
                    creditLoyalty(apdu);
195                 break;

                // transakcja
                case CMD_SUB_POINTS:
```

```
                    debitLoyalty(apdu);
                    break;

                // weryfikacja kodu PIN
                case CMD_VERIFY_PIN:
                    verifyPIN(apdu);
                    break;

                // generacja liczby pseudolosowej
                case CMD_ASK_RANDOM:
                    askRandom(apdu);
                    break;

                // uwierzytelnienie karty
                case CMD_INT_AUTH:
                    internalAuth(apdu);
                    break;

                // uwierzytelnienie terminala
                case CMD_EXT_AUTH:
                    externalAuth(apdu);
                    break;

                // każda inna instrukcja jest niepoprawna
                default:
                    // zgłoszenie wyjątku
                    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
                    break;
            }
        }

        // zwraca aktualną ilość punktów w programie P1 (wartość 1 lub 2)
        // lub sumę zebranych punktów P1 = 3
        private void getPoints(APDU apdu)
        {
            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();

            // wpisanie aktualnej ilości punktów
            if (apduBuffer[ISO7816.OFFSET_P1] == 1 ||
                apduBuffer[ISO7816.OFFSET_P1] == 2)
            {
                // partner 1 lub 2
                apduBuffer[5] = (byte)(points[apduBuffer[ISO7816.OFFSET_P1]-1] >> 8);
                apduBuffer[6] = (byte)points[apduBuffer[ISO7816.OFFSET_P1]-1];
            }
            else
            if (apduBuffer[ISO7816.OFFSET_P1] == 3)
            {
                // suma punktów
                apduBuffer[5] = (byte)(allPoints >> 8);
                apduBuffer[6] = (byte)allPoints;
            }
            else
                // błędny parametr
                ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);

            // przejście w tryb odpowiedzi
            apdu.setOutgoing();
            // długość odpowiedzi
            apdu.setOutgoingLength((short)2);

            // przesłanie odpowiedzi APDU (2 bajty od bajtu 5 z bufora)
            apdu.sendBytes((short)5, (short)2);
        }

        // dodanie punktów dla partnera P1 (wartość 1 lub 2)
        private void creditLoyalty(APDU apdu) throws ISOException
        {
```

```
            // terminal nie był uwierzytelniony
            if (!externalAuth)
                ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();

            // sprawdzenie poprawności numeru partnera
            if (apduBuffer[ISO7816.OFFSET_P1] != 1 &&
                apduBuffer[ISO7816.OFFSET_P1] != 2)
                ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);

            // sprawdzenie czy dostarczono 2 bajty danych (ilość punktów)
            if( apduBuffer[4] != 2 || apdu.setIncomingAndReceive() != 2 )
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

            // odczytanie wartości doładowania (bajty 5 i 6 w przesłaniej komendzie APDU)
            short newPoints = (short)(((apduBuffer[5] << 8 ) & (short)0xFF00) +
                              ((apduBuffer[6]) & (short)0x00FF));

            // spawdzenie poprawności wartości doładowania
            if ((newPoints>(short)(maxPoints-points[apduBuffer[ISO7816.OFFSET_P1]-1]))
                    || (newPoints <= (short)0))
                ISOException.throwIt(ISO7816.SW_DATA_INVALID);
            else
            {
                // doładowanie punktów
                points[apduBuffer[ISO7816.OFFSET_P1]-1] += newPoints;
                allPoints += newPoints;
            }
        }

        // wypłata gotówki
        private void debitLoyalty(APDU apdu) throws ISOException
        {
            // sprawdzenie czy wcześniej podano prawidłowy kod PIN i czy terminal był uwierzytelniony
            if (!ownerPIN.isValidated() || !externalAuth)
                ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();

            // sprawdzenie poprawności numeru partnera
            if (apduBuffer[ISO7816.OFFSET_P1] != 1 &&
                apduBuffer[ISO7816.OFFSET_P1] != 2)
                ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);

            // sprawdzenie czy dostarczono 2 bajty danych (ilość punktów) i czy udało się je odebrać
            if(apduBuffer[4] != 2 || apdu.setIncomingAndReceive() != 2)
                // operacja zakończona z błędem
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;

            // odczytanie wartości transakcji (bajty 5 i 6 w przesłaniej komendzie APDU)
            short newPoints = (short)(((apduBuffer[5] << 8 ) & (short)0xFF00) +
                              ((apduBuffer[6]) & (short)0x00FF));

            // sprawdzenie czy ilość punktów jest poprawna
            if ((points[apduBuffer[ISO7816.OFFSET_P1]-1] >= newPoints)
                    && (newPoints > 0))
            {
                // uaktualnienie stanu programu danego partnera
                points[apduBuffer[ISO7816.OFFSET_P1]-1] -= newPoints ;

                // nowy stan programu zostanie przesłany w odpowiedzi
                apduBuffer[9] = (byte)(points[apduBuffer[ISO7816.OFFSET_P1]-1] >> 8);
                apduBuffer[10] = (byte)points[apduBuffer[ISO7816.OFFSET_P1]-1];

                // przesłanie 2 bajtów odpowiedzi
                apdu.setOutgoing();
```

```
            apdu.setOutgoingLength((short)2);
            apdu.sendBytes((short)9, (short)2);
        }
        else
            // ilość punktów po operacji nie byłaby poprawna
            throw new ISOException (ISO7816.SW_DATA_INVALID) ;
    }

    // weryfikacja kodu PIN
    private void verifyPIN(APDU apdu)
    {
        if (!externalAuth)
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        // pobranie bufora z APDU
        byte[] buffer = apdu.getBuffer();

        // długość kodu PIN
        byte pinLength = buffer[ISO7816.OFFSET_LC];

        // pobranie przekazanych danych
        if(apdu.setIncomingAndReceive()<pinLength)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        if(!ownerPIN.check(buffer, ISO7816.OFFSET_CDATA, pinLength))
        {
            // sprawdzenie kodu PIN nie powiodło się
            byte triesRemaining = ownerPIN.getTriesRemaining();
            // ilość pozostałych prób wynosi x (63 Cx)
            ISOException.throwIt((short)(SW_WRONG_PIN + triesRemaining));
        }
    }

    // zwraca liczbę losową (8 bajtów)
    private void askRandom(APDU apdu)
    {
        // pobranie bufora z APDU
        byte[] apduBuffer = apdu.getBuffer();

        // wygenerowanie liczby losowej (za komendą APDU)
        random.generateData(apduBuffer, (short)5, (short)8);

        // skopiowanie liczby losowej do bufora
        Util.arrayCopy(apduBuffer, (short)5, lastRandom, (short)0, (short)8);

        // ustawienie flagi oznaczającej wygenerowanie liczby
        randomGen=true;

        // przejście w tryb odpowiedzi
        apdu.setOutgoing();
        // długość odpowiedzi
        apdu.setOutgoingLength((short)8);

        // przesłanie odpowiedzi APDU (8 bajtów od bajtu 5 z bufora)
        apdu.sendBytes((short)5, (short)8);
    }

    // uwierzytelnienie karty
    private void internalAuth(APDU apdu)
    {
        // pobranie bufora z APDU
        byte[] apduBuffer = apdu.getBuffer();

        // oczekujemy 8 bajtów danych
        if(apdu.setIncomingAndReceive() != 8)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        // ustawienie w tryb szyfrowania
        cipherDES.init(desKey, Cipher.MODE_ENCRYPT);

```

```
            // szyfrowanie
            cipherDES.doFinal(apduBuffer, (short)5, (short)8, apduBuffer, (short)13);

            // przejście w tryb odpowiedzi
            apdu.setOutgoing();
            // długość odpowiedzi
            apdu.setOutgoingLength((short)8);

            // przesłanie odpowiedzi APDU (8 bajty od bajtu 13 z bufora)
            apdu.sendBytes((short)13, (short)8);
        }

        // uwierzytelnienie terminala
        private void externalAuth(APDU apdu)
        {
            // nie było zapytania o liczbę losową
            if (!randomGen)
                ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

            // pobranie bufora z APDU
            byte[] apduBuffer = apdu.getBuffer();

            // oczekujemy 8 bajtów szyfrogramu
            if(apdu.setIncomingAndReceive() != 8)
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

            // tymczasowy bufor
            byte[] buffer = new byte[8];

            // liczba losowa już nie jest ważna (ochrona przed atakiem)
            randomGen=false;

            // deszyfrowanie
            cipherDES.init(desKey, Cipher.MODE_DECRYPT);
            cipherDES.doFinal(apduBuffer, (short)5, (short)8, buffer, (short)0);

            // jeśli po zdeszyfrowaniu szyfrogramu nie otrzymano ostatniej
            // liczby losowej - operacja nie powiodła się
            if (Util.arrayCompare(buffer, (short)0, lastRandom,
                                  (short)0, (short)8) != (short)0)
                ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
            else
                // w przeciwnym przypadku terminal jest uwierzytelniony
                externalAuth=true;
        }
    }
```

Wydruk 34. Aplet lojalnościowy (`LoyaltyExample.java`)

Uwagi bibliograficzne

Kilka informacji o aplikacjach lojalnościowych znajduje się w [3].

10. Infrastruktura klucza publicznego

W infrastrukturze klucza publicznego (PKI, ang. *public key infrastructure*) karty inteligentne wykorzystywane są jako nośnik wrażliwej informacji jaką jest klucz prywatny właściciela. Specjalnie dostosowany system operacyjny takich kart pozwala na wygenerowanie i operacje na kluczu prywatnym dzięki czemu nigdy nie opuszcza on bezpiecznego środowiska.

Poniższy rozdział dostarcza podstawowe informacje dotyczące zasad działania infrastruktury klucza publicznego ze szczególnym uwzględnieniem standardów przemysłowych. Jako przykład omówiony został pakiet narzędzi *OpenSC*.

10.1. Działanie systemu

Pełne wykorzystanie możliwości jakie stwarza podpis elektroniczny polega nie tylko na wdrożeniu aplikacji i komponentów umożliwiających jego składanie ale również jego weryfikację. Aby zrealizować ten cel należy zbudować infrastrukturę klucza publicznego. Składają się na nią następujące elementy:

— pary kluczy, w które wyposażeni są użytkownicy systemu (podpis realizowany jest w oparciu o algorytmy asymetryczne, w których wykorzystywane są dwa klucze: publiczny (jawny) i prywatny),
— certyfikaty, które wiążą klucz publiczny z danymi pozwalającymi jednoznacznie określić tożsamość właściciela (jest to rodzaj swoistego elektronicznego dowodu osobistego),
— zaufana trzecia strona, która jest odpowiedzialna za wystawianie certyfikatów dla użytkowników systemu (zakładamy, że ufają oni temu organowi, dzięki czemu, na podstawie wystawionych przez niego certyfikatów, są w stanie weryfikować swoją tożsamość), wystawienie certyfikatu polega na podpisaniu danych użytkownika (w tym jego klucza publicznego) przez organ certyfikujący,
— usługi katalogowe, które pozwalają na dystrybucję certyfikatów,
— dodatkowe usługi takie jak: serwery znakowania czasem, serwery przechowujące podpisane dokumenty, serwery CRL.

Oprócz wymienionych elementów równie ważnym składnikiem PKI są procedury formalne (np. przy wystawieniu lub unieważnieniu certyfikatu).

W infrastrukturze klucza publicznego karta procesorowa może być odpowiedzialna za:

— wygenerowanie pary kluczy kryptograficznych,
— operacje kryptograficzne z użyciem klucza prywatnego przeprowadzane w taki sposób by nie opuszczał on karty,
— ochronę przed nieuprawnionym dostępem do klucza prywatnego (kody PIN),
— przechowywanie certyfikatów (tak by użytkownik i oprogramowanie z którego korzysta miało do nich stały dostęp).

Tak spreparowana karta we współpracy z odpowiednim oprogramowaniem wykorzystywana jest do składania podpisów cyfrowych na dokumentach. Operacja złożenia podpisu przebiega najczęściej w następujący sposób:

— aplikacja oblicza wartość funkcji skrótu dokumentu, który ma być podpisany,
— użytkownik jest uwierzytelniany względem karty (np. poprzez podanie kodu PIN),
— wartość funkcji skrótu przesyłana jest do karty gdzie jest szyfrowana kluczem prywatnym,
— wynik tej operacji jest scalany z podpisywanym dokumentem.

Weryfikacja prawdziwości złożonego podpisu polega na wykonaniu następujących czynności:

— osoba weryfikująca podpis cyfrowy pobiera certyfikat osoby, która ten podpis złożyła,

— kluczem publicznym wystawcy certyfikatu odszyfrowana jest wartość funkcji skrótu podpisanego dokumentu,
— niezależnie licznona jest wartość funkcji skrótu dokumentu,
— wyznaczone wartości funkcji skrótu są porównywane - w przypadku otrzymania identycznych wartości przyjmujemy, że podpis złożyła osoba związana z pobranym certyfikatem.

Aby powyższe procedury mogły być wykorzystywane przez szeroką rzeszę użytkowników potrzebna jest standaryzacja formatów w jakim przechowywane są np. certyfikaty lub podpisy cyfrowe. W przeciwnym wypadku nie jest możliwa poprawna interpretacja wymienianych danych. Jednym z przykładów normalizacji w tym zakresie są zalecenia z serii PKCS opisane w kolejnym podrozdziale.

Podpis elektroniczny nie jest jedyną możliwością wykorzystania infrastruktury klucza publicznego. Na bazie certyfikatów można zrealizować możliwość uwierzytelnienia użytkowników w systemach komputerowych lub regulować dostęp do pewnych usług. Wygenerowanie pary kluczy dedykowanej do szyfrowania danych pozwala na bezpieczną komunikację poprzez pocztę elektroniczną lub w ramach sieci wirtualnej. Do tego celu jest wykorzystywana ta sama infrastruktura. Trzeba również zauważyć, że PKI jest w pełni skalowalne tj. organy certyfikujące mogą podlegać instytucjom wyższym w tej hierarchii i jest to przezroczyste dla użytkowników końcowych.

10.2. PKCS

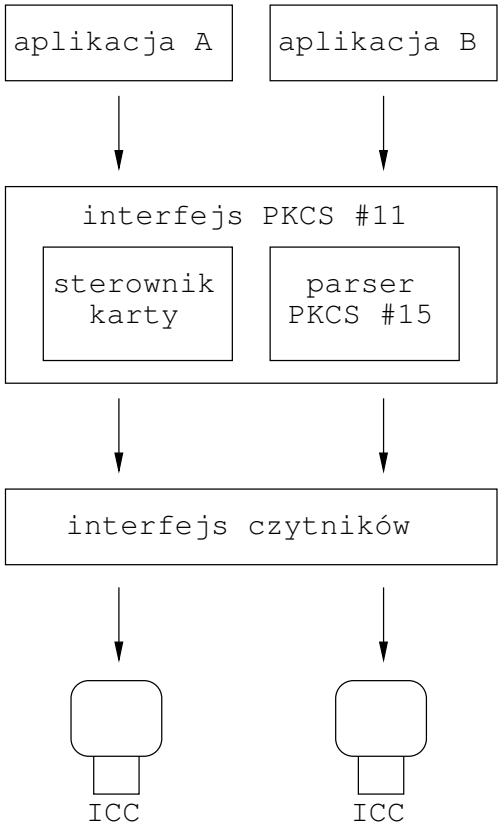
Firma *RSA Data Security, Inc.* zaproponowała szereg zaleceń dotyczących interfejsu kryptografii z kluczem publicznych. Znane są one pod ogólną nazwą PKCS (ang. *Public Key Cryptography Standards*). Mimo iż dokumenty te nie są normami (standardami) w pełnym znaczeniu tego słowa (wydawane i edytowane są przez komercyjną instytucję) to przyjęły się w świecie oprogramowania kryptograficznego.

Lista dokumentów z serii PKCS jest następująca:
— PKCS #1: *RSA Cryptography Standard* – zawiera opis algorytmu RSA zarówno w odniesieniu do podpisu cyfrowego jak i kopert cyfrowych,
— PKCS #3: *Diffie-Hellman Key Agreement Standard* – opisuje sposób implementacji algorytmu uzgadniania kluczy metodą Diffiego-Hellmana,
— PKCS #5: *Password-Based Cryptography Standard* – zawiera opis metody bezpiecznej wymiany kluczy prywatnych,
— PKCS #6: *Extended-Certificate Syntax Standard* – opisuje budowę certyfikatów klucza publicznego X.509,
— PKCS #7: *Cryptographic Message Syntax Standard* – jest to abstrakcyjny opis danych, które podlegają operacjom kryptograficznym ,
— PKCS #8: *Private-Key Information Syntax Standard* – zawiera abstrakcyjny opis dotyczący składowania kluczy prywatnych (w formie jawnej i zaszyfrowanej) wraz z zestawem atrybutów,
— PKCS #9: *Selected Attribute Types* – zawiera definicję atrybutów związanych z certyfikatami, podpisami cyfrowymi i kluczami prywatnymi,
— PKCS #10: *Certification Request Syntax Standard* – opisuje format żądania certyfikacyjnego,
— PKCS #11: *Cryptographic Token Interface Standard* – opisuje abstrakcyjny interfejs programisty dla różnych typów urządzeń kryptograficznych,
— PKCS #12: *Personal Information Exchange Syntax Standard* – zawiera opis formatu zapisu danych kryptograficznych przez aplikacje,
— PKCS #13: *Elliptic Curve Cryptography Standard* – zawiera opis algorytmów opartych na krzywych eliptycznych,

— PKCS #14: *Pseudo Random Number Generation* – zawiera opis algorytmów związanych z generacją liczb pseudolosowych[16],
— PKCS #15: *Cryptographic Token Information Format Standard* – opisuje sposób zapisu danych w tokenach kryptograficznych (takich jak karty procesorowe).

Wszystkie publikacje dostępne są pod adresem `http://www.rsasecurity.com`.

Z kartami inteligentnymi najmocniej związane są dokumenty PKCS #15 oraz PKCS #11, które dokładniej zostały opisane w kolejnych podrozdziałach. Zależność pomiędzy tymi zaleceniami pokazano na rysunku 26.

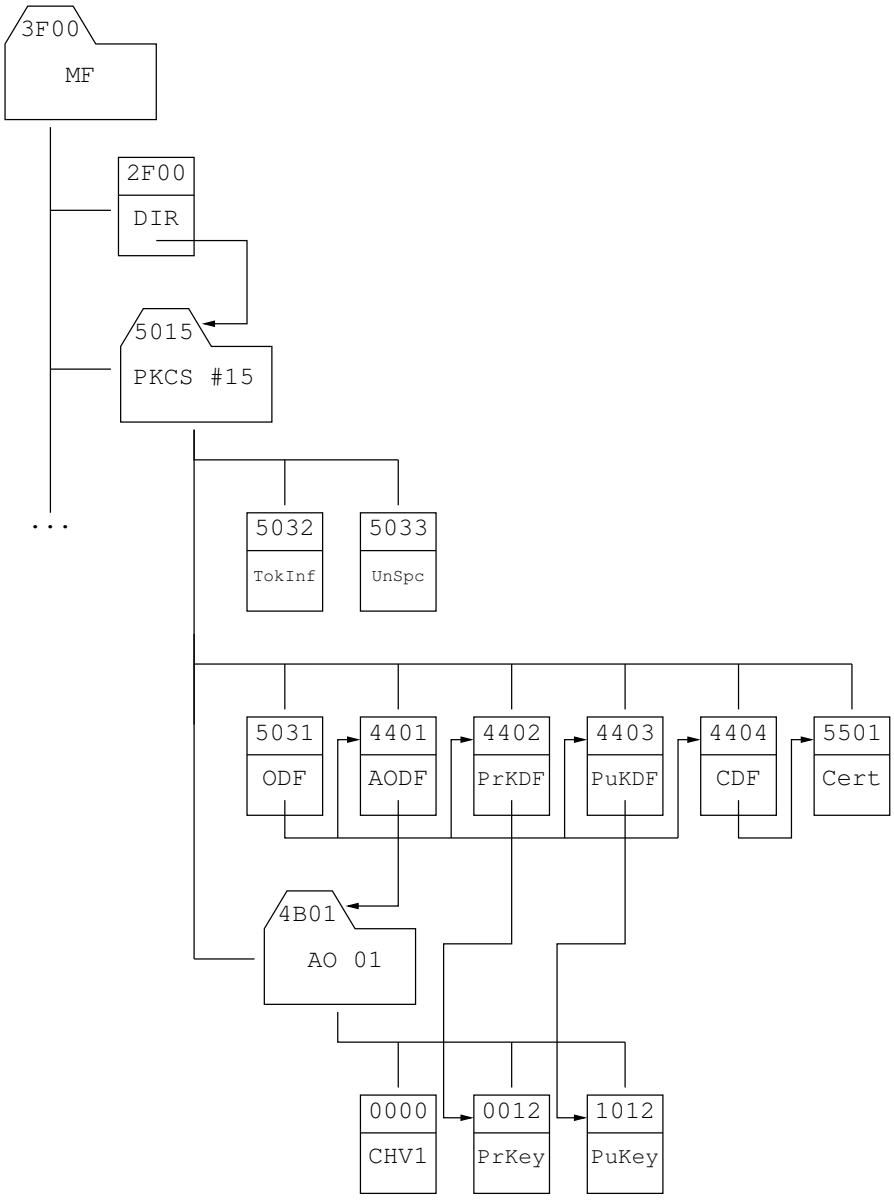


Rysunek 26. Zależność pomiędzy PKCS #11 i PKCS #15

10.2.1. PKCS #15

Zalecenia PKCS #15 definiują model informacji związanych z PKI w urządzeniach przeznaczonych do zarządzania nimi (tzw. tokenach), którymi w szczególności mogą być karty procesorowe. Nie zdefiniowano tu określonych komend jakie służą do wykonywania pewnych operacji (np. odczytu pliku itp.). Nie zdefiniowano również wysokopoziomowego interfejsu zarządzania tymi danymi. Opis dotyczy warstwy pośredniej, która odpowiedzialna jest za interpretację oraz lokalizację danych w określonych strukturach (systemie plików karty). Dlatego też w wymaganiach dotyczących karty określono, że powinna ona wspierać niezbędne funkcjonalności takie jak: hierarchiczny system plików i możliwość zarządzania nim, zarządzanie dostępem do informacji, możliwość wyboru aplikacji poprzez nazwę lub identyfikator katalogu, zaawansowane zarządzanie bezpieczeństwem (np. zabezpieczona komunikacja z kartą).

[16] aktualnie dokument ten jest opracowywany



Rysunek 27. Przykładowa struktura aplikacji PKCS #15

Hierarchia obiektów zdefiniowanych w PKCS #15 zdefiniowana jest w następujący sposób:
- główny obiekt PKCS #15 (ang. *PKCS #15 Top Object*) – zdefiniowany jako abstrakcyjny obiekt składający się z obiektów wybranych z czterech typów:
  - klucz (ang. *Key Object*) – obiekt ten reprezentuje klucz kryptograficzny:
    - klucz prywatny (ang. *Private Key*) – klucz tajny (dla algorytmów asymetrycznych np. RSA lub opartych na krzywych eliptycznych),
    - klucz publiczny (ang. *Public Key*) – klucz jawny (dla algorytmów asymetrycznych np. RSA lub opartych na krzywych eliptycznych),
    - klucz tajny (ang. *Secret Key*) – klucz kryptograficzny (dla algorytmów symetrycznych np. DES);
  - certyfikat (ang. *Certificate Object*) – typ danych reprezentujący certyfikat:
    - certyfikat X.509 (ang. *X.509 Certificate*) – certyfikat w formacie X.509,
    - inny certyfikat (ang. *Other Certificates*) – certyfikat w formacie innym niż X.509;
  - obiekt danych (ang. *Data Object*) – reprezentuje obiekty inne niż klucze i certyfikaty:
    - zewnętrzny obiekt danych (ang. *External Data Objects*) – obiekt przeznaczony do przechowywania danych związanych z aplikacją PKCS #15;
  - obiekt służący do uwierzytelnienia (ang. *Authentication Object*) – zawierają dane służące do weryfikacji tożsamości użytkownika w ramach aplikacji PKCS #15:
    - kody PIN (ang. *PIN Object*) – reprezentuje hasło użytkownika,
    - dane biometryczne (ang. *Biometric Template*) – pozwalają uwierzytelnić użytkownika np. na podstawie informacji o odcisku palca lub budowie tęczówki oka.

Przykładowa struktura karty z aplikacją zgodną z PKCS #15 jest zaprezentowana na rysunku 27. Kolejne elementy systemu plików odzwierciedlają instancje obiektów z danymi zdefiniowanymi w PKCS #15. Ich szczegółowa budowa, określona z użyciem notacji ASN.1 (ang. *abstract syntax notation one*) przedstawiona jest w samej normie.

Fizyczny dostęp do określonych obiektów (operacje związane z ich tworzeniem, odczytem, usuwaniem, modyfikacją) może być realizowany na kilku poziomach bezpieczeństwa:
- zabroniony (operacja nie jest możliwa),
- dostępny (operacja jest możliwa bez spełnienia dodatkowych warunków),
- po uwierzytelnieniu użytkownika,
- po uwierzytelnieniu zarządcy karty (wydawcy karty).

Nawiązując do przykładowej struktury, obiekty na niej zdefiniowane mają następujące przeznaczenie:
- MF (ang. *Master File*) – główny katalog w karcie, zazwyczaj tworzony przez dostawcę,
- DIR (ang. *Directory File*) – opcjonalny plik w głównym katalogu karty; zawiera informacje dotyczące aplikacji zawierające ich AID skojarzone z odpowiednimi identyfikatorami katalogów (ścieżkami),
- PKCS #15 (ang. *PKCS #15 Directory*) – katalog w którym umieszczono pliki aplikacji PKCS #15,
- ODF (ang. *Object Directory File*) – obowiązkowy obiekt w aplikacji PKCS #15 zawierający wskaźniki do innych plików elementarnych (z kluczami, certyfikatami itp.),
- AODF (ang. *Authentication Object Directory File*) – plik elementarny zawierający wskazania do obiektów wykorzystywanych przy uwierzytelnianiu użytkownika karty,
- PrKDF (ang. *Private Key Directory File*) – plik elementarny zawierający wskazania do obiektów będących kluczami prywatnymi,
- PuKDF (ang. *Public Key Directory File*) – plik elementarny zawierający wskazania do obiektów będących kluczami jawnymi,

— CDF (ang. *Certificate Directory File*) – plik elementarny zawierający wskazania do obiektów reprezentujących certyfikaty,
— Cert (ang. *Certificate*) – certyfikat,
— TokInf (ang. *Token Info*) – obowiązkowy plik elementarny, który zwiera informacje dotyczące tokenu m. in. jego numer seryjny, wspierane algorytmy kryptograficzne, identyfikatory wydawcy oraz właściciela karty itp.,
— UnSpc (ang. *Unused Space*) – opcjonalny plik elementarny, który zwiera informacje związane z wolnym miejscem na dane w innych plikach elementarnych,
— AO 01 (ang. *Authentication Object*) – katalog zawierający obiekty związane z uwierzytelnieniem,
— CHV1 (ang. *Cardholder Verify*) – kod PIN użytkownika,
— PrKey (ang. *Private Key*) – klucz prywatny, którego użycie chronione jest kodem PIN,
— PuKey (ang. *Public Key*) – klucz publiczny.

Zarządzanie strukturą danych w karcie związane jest z operacjami tworzenia nowych obiektów, ich modyfikowania oraz usuwania. W większości przypadków, podczas wstępnej personalizacji, należy z góry założyć ilość oraz wielkość plików z danymi. Zarządzanie samą strukturą oparte jest na modyfikacji odpowiednich wskazań do plików. Przykładowo w przypadku usuwania certyfikatu plik, w którym był zapisany, wypełniany jest zerami. Zwolniony obszar rejestrowany jest w odpowiednim pliku. Dzięki takim rozwiązaniom możliwe jest wykorzystanie kart, które nie obsługują komend usuwania fizycznego plików. Wadą jest konieczność założenia rozmiarów danych podczas tworzenia pierwotnej struktury PKCS #15. Trzeba jednak zauważyć, że taki specyficzny rodzaj zarządzania plikami jest charakterystyczny dla kart procesorowych.

W standardzie PKCS #15 zdefiniowano również wymagania wobec tzw. wirtualnej karty czyli tokena zrealizowanego w sposób programowy.

10.2.2. PKCS #11

Dokument PKCS #11 definiuje interfejs programisty (zwany „Cryptoki” od ang. *cryptographic token interface*) umożliwiający logiczny dostęp do tokenów. Tokenem nazywamy element (może być to np. urządzenie), który przechowuje dane i ma możliwość wykonywania operacji kryptograficznych. Danymi mogą być różnorakie klucze kryptograficzne, certyfikaty oraz obiekty innego typu, niepowiązane z kryptografią. Token ma możliwość tworzenia, modyfikacji oraz usuwania tych obiektów. Może wykonywać na nich lub z ich użyciem operacje kryptograficzne.

W warstwie PKCS #11 wyróżnia się dwóch użytkowników. Jednym z nich jest SO (ang. *security officer*), czyli zarządca. Odpowiedzialny jest on za inicjalizację tokenu (np. generacja kluczy kryptograficznych). Drugi z użytkowników staje się właścicielem tokenu. Nie ma on możliwości modyfikacji pewnych jego elementów np. kluczy kryptograficznych.

Tokeny umieszczone są w slotach. Aby zaistniała możliwość skorzystania z tokenu należy rozpocząć sesję. Warunkiem jej rozpoczęcia jest zalogowanie się (przedstawienie kodu PIN). W tym samym czasie wiele aplikacji może mieć rozpoczęte sesje z tym samym tokenem. Jednak w zależności od wykonywanych operacji (tylko odczyt, zapis) niektóre funkcje mogą być zablokowane.

Poniżej, w sposób bardzo ogólny, przedstawiony jest interfejs jaki zdefiniowano w „Cryptoki”. Celem jest jedynie zaprezentowanie możliwości, szczegółowa specyfikacja dostępna jest w [67].

Do podstawowych funkcji biblioteki zaliczyć można metody pozwalające na jej inicjalizację oraz zakończenie. Dodatkowe funkcje umożliwiają pobranie informacji o bibliotece a także

wskaźnik do listy funkcji jakie są zaimplementowane w danej bibliotece. Umożliwia to wykorzystanie kilku implementacji równolegle w jednym programie.

```
CK_RV C_Initialize(CK_VOID_PTR pInitArgs);
CK_RV C_Finalize(CK_VOID_PTR pReserved);
CK_RV C_GetInfo(CK_INFO_PTR pInfo);
CK_RV C_GetFunctionList(
            CK_FUNCTION_LIST_PTR_PTR ppFunctionList);
```

Po zainicjalizowaniu biblioteki możliwe jest wykorzystanie szeregu funkcji pozwalających na pobranie informacji o istniejących slotach, tokenach oraz operacjach jakie one umożliwiają. Do tego zestawu można zaliczyć również funkcje pozwalające na inicjalizację tokenu i jego kodów PIN. Ostatnia z funkcji pozwala na oczekiwanie na zdarzenia związane z tokenem (jego włożenie, usunięcie).

```
CK_RV C_GetSlotList(CK_BBOOL       tokenPresent,
                    CK_SLOT_ID_PTR pSlotList,
                    CK_ULONG_PTR   pulCount);
CK_RV C_GetSlotInfo(CK_SLOT_ID       slotID,
                    CK_SLOT_INFO_PTR pInfo);
CK_RV C_GetTokenInfo(CK_SLOT_ID        slotID,
                     CK_TOKEN_INFO_PTR pInfo);
CK_RV C_GetMechanismList(CK_SLOT_ID            slotID,
                         CK_MECHANISM_TYPE_PTR pMechanismList,
                         CK_ULONG_PTR          pulCount);
CK_RV C_GetMechanismInfo(CK_SLOT_ID            slotID,
                         CK_MECHANISM_TYPE     type,
                         CK_MECHANISM_INFO_PTR pInfo);
CK_RV C_InitToken(CK_SLOT_ID      slotID,
                  CK_UTF8CHAR_PTR pPin,
                  CK_ULONG        ulPinLen,
                  CK_UTF8CHAR_PTR pLabel);
CK_RV C_InitPIN(CK_SESSION_HANDLE hSession,
                CK_UTF8CHAR_PTR   pPin,
                CK_ULONG          ulPinLen);
CK_RV C_SetPIN(CK_SESSION_HANDLE hSession,
               CK_UTF8CHAR_PTR   pOldPin,
               CK_ULONG          ulOldLen,
               CK_UTF8CHAR_PTR   pNewPin,
               CK_ULONG          ulNewLen);
CK_RV C_WaitForSlotEvent(CK_FLAGS flags,
                         CK_SLOT_ID_PTR pSlot,
                         CK_VOID_PTR pRserved);
```

Aby możliwe było korzystanie z operacji jakich dostarcza dany token należy rozpocząć z nim sesję. Wiąże się to z operacją logowania do tokenu z użyciem kodu PIN. Po zakończeniu korzystania z tokenu należy zamknąć sesję. Dodatkowe funkcje pozwalają na pobranie kopii aktualnego stanu tokenu i jego późniejsze odtworzenie.

```
CK_RV C_OpenSession(CK_SLOT_ID            slotID,
                    CK_FLAGS              flags,
                    CK_VOID_PTR           pApplication,
                    CK_NOTIFY             Notify,
                    CK_SESSION_HANDLE_PTR phSession);
CK_RV C_CloseSession(CK_SESSION_HANDLE hSession);
CK_RV C_CloseAllSessions(CK_SLOT_ID      slotID);
CK_RV C_GetSessionInfo(CK_SESSION_HANDLE   hSession,
                       CK_SESSION_INFO_PTR pInfo);
CK_RV C_GetOperationState(
```

```
                    CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pOperationState,
                    CK_ULONG_PTR      pulOperationStateLen);
CK_RV C_SetOperationState(CK_SESSION_HANDLE hSession,
                          CK_BYTE_PTR       pOperationState,
                          CK_ULONG          ulOperationStateLen,
                          CK_OBJECT_HANDLE  hEncryptionKey,
                          CK_OBJECT_HANDLE  hAuthenticationKey);
CK_RV C_Login(CK_SESSION_HANDLE hSession,
              CK_USER_TYPE      userType,
              CK_UTF8CHAR_PTR   pPin,
              CK_ULONG          ulPinLen);
CK_RV C_Logout(CK_SESSION_HANDLE hSession);
```

Operacje związane z obiektami (poza kluczami) pozwalające na ich tworzenie, kopiowanie, usuwanie, pobieranie rozmiarów, ustalanie i pobieranie atrybutów oraz wyszukiwanie realizowane są z wykorzystaniem poniższego zestawu funkcji. Wyszukiwanie obiektów wiąże się z inicjalizacją tego procesu, szeregiem wywołań funkcji znajdującej kolejne obiekty i zakończeniem operacji wyszukiwania.

```
CK_RV C_CreateObject(CK_SESSION_HANDLE hSession,
                     CK_ATTRIBUTE_PTR  pTemplate,
                     CK_ULONG          ulCount,
                     CK_OBJECT_HANDLE_PTR phObject);
CK_RV C_CopyObject(CK_SESSION_HANDLE    hSession,
                   CK_OBJECT_HANDLE     hObject,
                   CK_ATTRIBUTE_PTR     pTemplate,
                   CK_ULONG             ulCount,
                   CK_OBJECT_HANDLE_PTR phNewObject);
CK_RV C_DestroyObject(CK_SESSION_HANDLE hSession,
                      CK_OBJECT_HANDLE  hObject);
CK_RV C_GetObjectSize(CK_SESSION_HANDLE hSession,
                      CK_OBJECT_HANDLE  hObject,
                      CK_ULONG_PTR      pulSize);
CK_RV C_GetAttributeValue(CK_SESSION_HANDLE hSession,
                          CK_OBJECT_HANDLE  hObject,
                          CK_ATTRIBUTE_PTR  pTemplate,
                          CK_ULONG          ulCount);
CK_RV C_SetAttributeValue(CK_SESSION_HANDLE hSession,
                          CK_OBJECT_HANDLE  hObject,
                          CK_ATTRIBUTE_PTR  pTemplate,
                          CK_ULONG          ulCount);
CK_RV C_FindObjectsInit(CK_SESSION_HANDLE hSession,
                        CK_ATTRIBUTE_PTR  pTemplate,
                        CK_ULONG          ulCount);
CK_RV C_FindObjects(CK_SESSION_HANDLE    hSession,
                    CK_OBJECT_HANDLE_PTR phObject,
                    CK_ULONG             ulMaxObjectCount,
                    CK_ULONG_PTR         pulObjectCount);
CK_RV C_FindObjectsFinal(CK_SESSION_HANDLE hSession);
```

Operacja szyfrowania związana jest z inicjalizacją tego procesu, a następnie skorzystaniem z funkcji szyfrującej całość lub pojedynczy element danych z bufora. Zakończenie operacji wymaga wywołania ostatniej z wymienionych poniżej funkcji.

```
CK_RV C_EncryptInit(CK_SESSION_HANDLE hSession,
                    CK_MECHANISM_PTR  pMechanism,
                    CK_OBJECT_HANDLE  hKey);
CK_RV C_Encrypt(CK_SESSION_HANDLE hSession,
```

```
                    CK_BYTE_PTR       pData,
                    CK_ULONG          ulDataLen,
                    CK_BYTE_PTR       pEncryptedData,
                    CK_ULONG_PTR      pulEncryptedDataLen);
CK_RV C_EncryptUpdate(CK_SESSION_HANDLE hSession,
                      CK_BYTE_PTR       pPart,
                      CK_ULONG          ulPartLen,
                      CK_BYTE_PTR       pEncryptedPart,
                      CK_ULONG_PTR      pulEncryptedPartLen);
CK_RV C_EncryptFinal(
                    CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pLastEncryptedPart,
                    CK_ULONG_PTR      pulLastEncryptedPartLen);
```

Dla operacji deszyfrowania zdefiniowano analogiczny interfejs jak dla szyfrowania.

```
CK_RV C_DecryptInit(CK_SESSION_HANDLE hSession,
                    CK_MECHANISM_PTR  pMechanism,
                    CK_OBJECT_HANDLE  hKey);
CK_RV C_Decrypt(CK_SESSION_HANDLE hSession,
                CK_BYTE_PTR       pEncryptedData,
                CK_ULONG          ulEncryptedDataLen,
                CK_BYTE_PTR       pData,
                CK_ULONG_PTR      pulDataLen);
CK_RV C_DecryptUpdate(CK_SESSION_HANDLE hSession,
                      CK_BYTE_PTR       pEncryptedPart,
                      CK_ULONG          ulEncryptedPartLen,
                      CK_BYTE_PTR       pPart,
                      CK_ULONG_PTR      pulPartLen);
CK_RV C_DecryptFinal(CK_SESSION_HANDLE hSession,
                     CK_BYTE_PTR       pLastPart,
                     CK_ULONG_PTR      pulLastPartLen);
```

Możliwe jest również korzystanie z funkcji skrótu poprzez token kryptograficzny. Podobnie jak dla poprzednich operacji w przypadku obliczania wartości danej funkcji dla większej ilości danych należy skorzystać z odpowiedniej metody. Aby do algorytmu obliczającego funkcję skrótu przekazać klucz należy skorzystać z przedostatniej z wymienionych poniżej nagłówków.

```
CK_RV C_DigestInit(CK_SESSION_HANDLE hSession,
                   CK_MECHANISM_PTR  pMechanism);
CK_RV C_Digest(CK_SESSION_HANDLE hSession,
               CK_BYTE_PTR       pData,
               CK_ULONG          ulDataLen,
               CK_BYTE_PTR       pDigest,
               CK_ULONG_PTR      pulDigestLen);
CK_RV C_DigestUpdate(CK_SESSION_HANDLE hSession,
                     CK_BYTE_PTR       pPart,
                     CK_ULONG          ulPartLen);
CK_RV C_DigestKey(CK_SESSION_HANDLE hSession,
                  CK_OBJECT_HANDLE  hKey);
CK_RV C_DigestFinal(CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pDigest,
                    CK_ULONG_PTR      pulDigestLen);
```

Operacje związane z podpisem cyfrowym oraz obliczaniem MAC dla danych przebiegają podobnie jak szyfrowanie lub deszyfrowanie. Ostatnie dwie z wymienionych funkcji pozwalają na wykonanie podpisu, z którego wartości może zostać odzyskana podpisywana informacja.

```
CK_RV C_SignInit(CK_SESSION_HANDLE hSession,
                 CK_MECHANISM_PTR  pMechanism,
```

```
                    CK_OBJECT_HANDLE   hKey);
CK_RV C_Sign(CK_SESSION_HANDLE hSession,
             CK_BYTE_PTR       pData,
             CK_ULONG          ulDataLen,
             CK_BYTE_PTR       pSignature,
             CK_ULONG_PTR      pulSignatureLen);
CK_RV C_SignUpdate(CK_SESSION_HANDLE hSession,
                   CK_BYTE_PTR       pPart,
                   CK_ULONG          ulPartLen);
CK_RV C_SignFinal(CK_SESSION_HANDLE hSession,
                  CK_BYTE_PTR       pSignature,
                  CK_ULONG_PTR      pulSignatureLen);
CK_RV C_SignRecoverInit(CK_SESSION_HANDLE hSession,
                        CK_MECHANISM_PTR  pMechanism,
                        CK_OBJECT_HANDLE  hKey);
CK_RV C_SignRecover(CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pData,
                    CK_ULONG          ulDataLen,
                    CK_BYTE_PTR       pSignature,
                    CK_ULONG_PTR      pulSignatureLen);
```

Odzwierciedleniem dla operacji z poprzedniego akapitu są funkcje dokonujące weryfikacji podpisu cyfrowego lub MAC. Ostatnie dwie metody przeznaczone są dla podpisów z których można odzyskać podpisywaną informację.

```
CK_RV C_VerifyInit(CK_SESSION_HANDLE hSession,
                   CK_MECHANISM_PTR  pMechanism,
                   CK_OBJECT_HANDLE  hKey);
CK_RV C_Verify(CK_SESSION_HANDLE hSession,
               CK_BYTE_PTR       pData,
               CK_ULONG          ulDataLen,
               CK_BYTE_PTR       pSignature,
               CK_ULONG          ulSignatureLen);
CK_RV C_VerifyUpdate(CK_SESSION_HANDLE hSession,
                     CK_BYTE_PTR       pPart,
                     CK_ULONG          ulPartLen);
CK_RV C_VerifyFinal(CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pSignature,
                    CK_ULONG          ulSignatureLen);
CK_RV C_VerifyRecoverInit(CK_SESSION_HANDLE hSession,
                          CK_MECHANISM_PTR  pMechanism,
                          CK_OBJECT_HANDLE  hKey);
CK_RV C_VerifyRecover(CK_SESSION_HANDLE hSession,
                      CK_BYTE_PTR       pSignature,
                      CK_ULONG          ulSignatureLen,
                      CK_BYTE_PTR       pData,
                      CK_ULONG_PTR      pulDataLen);
```

Często zachodzi potrzeba wykonania dwóch operacji bezpośrednio po sobie np. zaszyfrowania wartości obliczonej funkcji skrótu (oraz operacji odwrotnej). Aby uniknąć dwukrotnego przesyłania danych z i do tokenu przygotowano specjalne funkcje umożliwiające tego typu operacje.

```
CK_RV C_DigestEncryptUpdate(
                    CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pPart,
                    CK_ULONG          ulPartLen,
                    CK_BYTE_PTR       pEncryptedPart,
                    CK_ULONG_PTR      pulEncryptedPartLen);
CK_RV C_DecryptDigestUpdate(
```

```
                    CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pEncryptedPart,
                    CK_ULONG          ulEncryptedPartLen,
                    CK_BYTE_PTR       pPart,
                    CK_ULONG_PTR      pulPartLen);
CK_RV C_SignEncryptUpdate(
                    CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pPart,
                    CK_ULONG          ulPartLen,
                    CK_BYTE_PTR       pEncryptedPart,
                    CK_ULONG_PTR      pulEncryptedPartLen);
CK_RV C_DecryptVerifyUpdate(
                    CK_SESSION_HANDLE hSession,
                    CK_BYTE_PTR       pEncryptedPart,
                    CK_ULONG          ulEncryptedPartLen,
                    CK_BYTE_PTR       pPart,
                    CK_ULONG_PTR      pulPartLen);
```

Dla obiektów będących kluczami kryptograficznymi zdefiniowano funkcje umożliwiające ich generację (dla kluczy w algorytmach symetrycznych i asymetrycznych), szyfrowanie, deszyfrowanie oraz dywersyfikację. Operacje te pozwalają wykonywać działania na kluczach w bezpiecznym środowisku tokenu.

```
CK_RV C_GenerateKey(CK_SESSION_HANDLE     hSession,
                    CK_MECHANISM_PTR      pMechanism,
                    CK_ATTRIBUTE_PTR      pTemplate,
                    CK_ULONG              ulCount,
                    CK_OBJECT_HANDLE_PTR phKey);
CK_RV C_GenerateKeyPair(
              CK_SESSION_HANDLE     hSession,
              CK_MECHANISM_PTR      pMechanism,
              CK_ATTRIBUTE_PTR      pPublicKeyTemplate,
              CK_ULONG              ulPublicKeyAttributeCount,
              CK_ATTRIBUTE_PTR      pPrivateKeyTemplate,
              CK_ULONG              ulPrivateKeyAttributeCount,
              CK_OBJECT_HANDLE_PTR phPublicKey,
              CK_OBJECT_HANDLE_PTR phPrivateKey);
CK_RV C_WrapKey(CK_SESSION_HANDLE hSession,
                CK_MECHANISM_PTR  pMechanism,
                CK_OBJECT_HANDLE  hWrappingKey,
                CK_OBJECT_HANDLE  hKey,
                CK_BYTE_PTR       pWrappedKey,
                CK_ULONG_PTR      pulWrappedKeyLen);
CK_RV C_UnwrapKey(CK_SESSION_HANDLE     hSession,
                  CK_MECHANISM_PTR      pMechanism,
                  CK_OBJECT_HANDLE      hUnwrappingKey,
                  CK_BYTE_PTR           pWrappedKey,
                  CK_ULONG              ulWrappedKeyLen,
                  CK_ATTRIBUTE_PTR      pTemplate,
                  CK_ULONG              ulAttributeCount,
                  CK_OBJECT_HANDLE_PTR phKey);
CK_RV C_DeriveKey(CK_SESSION_HANDLE     hSession,
                  CK_MECHANISM_PTR      pMechanism,
                  CK_OBJECT_HANDLE      hBaseKey,
                  CK_ATTRIBUTE_PTR      pTemplate,
                  CK_ULONG              ulAttributeCount,
                  CK_OBJECT_HANDLE_PTR phKey);
```

Dostęp do generatora liczb pseudolosowych (lub losowych) w tokenie umożliwiają funkcje

pozwalające inicjalizować go z określonym ziarnem początkowym oraz pobierające losowe dane z tokena.

```
CK_RV C_SeedRandom(CK_SESSION_HANDLE hSession,
                   CK_BYTE_PTR       pSeed,
                   CK_ULONG          ulSeedLen);
CK_RV C_GenerateRandom(CK_SESSION_HANDLE hSession,
                       CK_BYTE_PTR       RandomData,
                       CK_ULONG          ulRandomLen);
```

Dla starszych implementacji PKCS #11 istniały funkcje pozwalające na zarządzanie funkcjami wykonywanymi równolegle (wątki). Obecnie nie są one wspierane. Służyły do monitorowania oraz ewentualnego zatrzymania operacji wykonywanej w tle.

```
CK_RV C_GetFunctionStatus(CK_SESSION_HANDLE hSessiond);
CK_RV C_CancelFunction(CK_SESSION_HANDLE hSession);
```

Wydruk 35 jest szkieletem aplikacji wykorzystującej implementację interfejsu PKCS #11. W pierwszej części znajdują się niezbędne definicje (zależne od platformy systemowej) jakich wymagają pliki nagłówkowe PKCS #11. Główna funkcja programu przedstawia kolejne kroki tj. inicjalizację biblioteki, otwarcie sesji, jej zamknięcie i zakończenie korzystania z modułu.

```
#include <stdio.h>
#include <assert.h>
#include <string.h>

// definicje zależne od platformy systemowej
#ifndef _WIN32

#define CK_PTR *

#define CK_DEFINE_FUNCTION(returnType, name) \
   returnType name

#define CK_DECLARE_FUNCTION(returnType, name) \
   returnType name

#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
   returnType (* name)

#define CK_CALLBACK_FUNCTION(returnType, name) \
   returnType (* name)

#ifndef NULL_PTR
#define NULL_PTR 0
#endif

#else

#define CK_PTR *

#define CK_DEFINE_FUNCTION(returnType, name) \
   returnType __declspec(dllexport) name

#define CK_DECLARE_FUNCTION(returnType, name) \
   returnType __declspec(dllexport) name

#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
   returnType __declspec(dllimport) (* name)

#define CK_CALLBACK_FUNCTION(returnType, name) \
   returnType (* name)

#ifndef NULL_PTR
#define NULL_PTR 0
#endif
```

```
45
#endif

#include "pkcs11.h"

50 int main()
{
    // wartość zwracana przez funkcje
    CK_RV rv;
    // sesja
55  CK_SESSION_HANDLE session = CK_INVALID_HANDLE;
    // obiekt
    CK_OBJECT_HANDLE object = CK_INVALID_HANDLE;
    // slot
    CK_SLOT_ID slot = 0;
60  // flagi
    CK_FLAGS flags = CKF_RW_SESSION | CKF_SERIAL_SESSION;

    // uchwyt do biblioteki
    CK_FUNCTION_LIST_PTR p11;
65
    // ładowanie biblioteki dynamicznej i pobranie uchwytu do listy funkcji
    // np. dla WIN32
    HINSTANCE hModule = LoadLibrary("pkcs11.dll");
    assert(hModule!=NULL);
70  (*GetProcAddress(hModule, "C_GetFunctionList"))(&p11);

    // inicjalizacja modułu
    rv = p11->C_Initialize(NULL);
    if (rv != CKR_OK)
75  {
        printf("C_Initialize␣error␣[%d]", rv);
        return 1;
    }

80  // pobranie informacji o slotach w systemie i ustalenie wartości
    // zmiennej slot
    // ...

    // otwarcie sesji
85  rv = p11->C_OpenSession(slot, flags, NULL, NULL, &session);
    if (rv != CKR_OK)
    {
        printf("C_OpenSession␣error␣[%d]", rv);
        return 1;
90  }

    // ...
    // operacje w czasie sesji
    // ...
95
    // zamknięcie sesji
    p11->C_CloseSession(session);

    // zakończenie korzystania z modułu
100 p11->C_Finalize(NULL_PTR);

    // wyładowanie biblioteki dynamicznej
    // ...

105 return 0;
};
```

Wydruk 35. Wykorzystanie interfejsu PKCS #11 (`PKCS11Example.c`)

10.3. OpenSC

OpenSC jest biblioteką wraz z zestawem aplikacji przeznaczoną dla kart będących tokenami. W szczególności uwzględniono karty umożliwiające stworzenie struktury PKCS #15.

Do najważniejszych funkcjonalności aplikacji i bibliotek można zaliczyć: możliwość operowania strukturą plików zgodną z ISO 7816-4, zarządzanie strukturą PKCS #15 oraz implementacje interfejsu PKCS #11. Aktualnie obsługiwanymi kartami lub systemami operacyjnymi są: *Cryptoflex*, *GPK*, *CardOS*, *Eutron CryptoIdendity IT-SEC*, *Micardo*, *Miocos*, *SetCOS*, *Tcos*. Ze względu na różnorodność obsługiwanych systemów operacyjnych specyficzne elementy PKCS #15 (takie jak prawa dostępu) wyszczególnione są w tzw. profilach, z których korzysta aplikacja inicjalizująca w karcie tą strukturę. Modyfikacje profilu pozwalają dostosować układ plików do wymagań dla konkretnej implementacji PKI. W pakiecie *OpenSC* zostały przygotowane przykładowe profile dla popularnych kart.

Aplikacje umożliwiają komunikowanie się poprzez interfejsy PC/SC (zobacz 6.2), CT-API (zobacz 6.1) oraz OpenCT (zobacz 6.3).

Kody źródłowe oraz wersje binarne (dla *Linux* i *Microsoft Windows*) wraz z dokumentacją dostępne są pod adresem `http://www.opensc.org`.

10.3.1. Narzędzia

Narzędzia wchodzące w skład pakietu *OpenSC* pozwalają wykorzystać główne funkcjonalności dostarczonych bibliotek.

Program *opensc-config* umożliwia sprawdzenie aktualnej instalacji pakietu w systemie oraz najważniejszych opcji konfiguracyjnych tej wersji.

Aplikacja *opensc-tool* pozwala na uzyskanie informacji o czytnikach zainstalowanych w systemie operacyjnym i wspieranych przez bibliotekę kartach. Możliwe jest również pozyskanie ATRkarty, przesyłanie komend APDU do karty w danym czytniku. Dla niektórych kart można wykonać skanowanie zawartości systemu plików.

Interaktywna aplikacja *opensc-explorer* pozwala na edycję zawartości systemu plików dla obsługiwanych kart. Możliwe jest skanowanie systemu plików, odczyt i zapis danych, tworzenie i usuwanie plików, a także operacje specyficzne dla danych kart jak np. modyfikacje kodów PIN i obsługa specyficznych obiektów dla danych systemów operacyjnych.

Program *pkcs15-init* ułatwia personalizację kart, które będą wykorzystywane jako tokeny kryptograficzne. Tworzone struktury danych w karcie mogą być modyfikowane poprzez edycję profili dla danych systemów operacyjnych. Aplikacja pozwala na tworzenie struktury plików, generację kluczy, zapis i odczyt certyfikatów, operacje na kodach PIN. W celu utworzenia przykładowej struktury można posłużyć się następującym ciągiem komend:

```
// wyczyszczenie karty oraz utworzenie podstawowej struktury
pkcs15-init -EC

// zapis kodu PIN w karcie
pkcs15-init -P  --auth-id 01 --pin 1234 --puk 4321 --label "Kod PIN"

// generacja w karcie klucza RSA chronionego kodem PIN
pkcs15-init -G rsa/1024 --auth-id 01

// zapis do karty certyfikatu wraz z kluczem (format PKCS #12)
pkcs15-init --auth-id 01 -f pkcs12 -S cert.p12
```

Aplikacja *pkcs15-tool* przeznaczona jest do manipulacji oraz uzyskiwania informacji o strukturze danych w karcie. Możliwy jest odczyt zasobów z karty (obiektów danych), listy kluczy i certyfikatów. Program pozwala także na zmianę kodów PIN.

Operacje kryptograficzne z użyciem karty o strukturze PKCS #15 umożliwia *pkcs15-crypt*. Do najważniejszych operacji można zaliczyć deszyfrowanie danych oraz podpis cyfrowy z użyciem kluczy przechowywanych w karcie.

Analogiczną, pod względem funkcjonalności, do poprzedniej aplikacją, ale wykorzystującą interfejs PKCS #11 jest *pkcs11-tool*. Z jej użyciem możliwe jest zarządzanie danymi w tokenach wyposażonych w interfejs PKCS #11. Użytkownik może sprawdzić listę kluczy, certyfikatów i kodów PIN w tokenie. Aplikacja pozwala również na sprawdzenie właściwości tokenu (obsługiwane algorytmy kryptograficzne). Możliwe są również operacje kryptograficzne np. podpis cyfrowy.

OpenSC zawiera również narzędzia dedykowane dla pewnych rodzajów kart. Zaliczyć do nich należy *cryptoflex-tool*, które pozwala na zarządzanie kartami z serii *Cryptoflex* oraz *cardos-info* przeznaczone dla kart wyposażonych w system *CardOS*.

Oprócz aplikacji pakiet *OpenSC* zawiera również moduł PKCS #11 dla obsługiwanych kart, który może być wykorzystany np. w przeglądarce internetowej lub programie pocztowym. Dla systemu *Linux* stworzony jest moduł PAM (ang. *Pluggable Authentication Module*), który umożliwia uwierzytelnienie użytkownika z użyciem karty i certyfikatu. W pakiecie są również wtyczki dla *OpenSSL* oraz *OpenSSH* pozwalające na wykorzystanie możliwości kart w tych aplikacjach.

10.3.2. Interfejs programisty

W rozdziale zaprezentowane są podstawowe elementu interfejsu dostarczanego przez bibliotekę *OpenSC*. Pozwalają one na proste operacje związane z dostępem do karty oraz modyfikacją jej systemu plików.

W celu inicjalizacji biblioteki oraz jej dezaktywacji należy skorzystać z następujących funkcji:

```
int sc_establish_context(sc_context_t **ctx,
                         const char   *app_name);
int sc_release_context(struct sc_context **ctx);
```

ctx kontekst połączenia, struktura zawiera pola określające dostępne czytniki w systemie oraz informacje o nich

app_name nazwa aplikacji; parametr ten pozwala skojarzyć ustawienia parametrów dla biblioteki dla danej aplikacji zgodnie z zapisami w pliku `opensc.conf`

wartość zwracana wartość ujemna oznacza błąd

Funkcja, która pozwala na sprawdzenie obecności karty w czytniku (wspierana tylko przez niektóre interfejsy komunikacyjne) to:

```
int sc_detect_card_presence(sc_reader_t *reader,
                            int          slot);
```

reader struktura reprezentująca czytnik

slot slot czytnika

wartość zwracana wartość ujemna w przypadku błędu lub wartość dodatnia, w której mogą wystąpić flagi:

— `SC_SLOT_CARD_PRESENT` – karta włożona do czytnika

— `SC_SLOT_CARD_CHANGED` – stan czytnika zmienił się od czasu ostatniego wywołania

Rozpoczęcie wykonywania operacji w karcie musi być poprzedzone wywołaniem pierwszej z poniższych funkcji. Po zakończeniu korzystania z karty, przy wykorzystaniu kolejnej funkcji, zwalniamy elementy struktury reprezentującej kartę.

```
int sc_connect_card(sc_reader_t  *reader,
                    int          slot,
                    sc_card_t    **card);
int sc_disconnect_card(sc_card_t *card,
                       int       action);
```

reader struktura reprezentująca czytnik
slot slot czytnika
card struktura reprezentująca podstawowe informacje o karcie (ATR, typ karty, biblioteka obsługująca kartę itp.)
action parametr aktualnie nieużywany, powinien być ustawiony na 0
wartość zwracana ujemna w przypadku błędu

Na blokadę karty (tak by nie mogły korzystać z niej inne aplikacje) oraz jej późniejsze odblokowanie pozwalają funkcje:

```
int sc_lock(sc_card_t *card);
int sc_unlock(sc_card_t *card);
```

card kontekst połączenia z kartą
wartość zwracana wartość ujemna w przypadku błędu

Odczytanie identyfikatorów plików występujących w aktualnie wybranym katalogu możliwe jest przy użyciu:

```
int sc_list_files(sc_card_t     *card,
                  unsigned char *buffer,
                  size_t        buflen);
```

card kontekst połączenia z kartą
buffer, buflen bufor na identyfikatory plików i wielkość tego bufora
wartość zwracana wartość ujemna w przypadku błędu

Obiekty systemu plików (katalogi, pliki elementarne) reprezentowane są w *OpenSC* przez odpowiednią strukturę.

```
typedef struct sc_file
{
    struct sc_path path;
    int            type, ef_structure;
    size_t         size;
    int            id;
    int            record_length;
    int            record_count;
} sc_file_t;
```

path ścieżka do obiektu (pełna)
type typ obiektu (katalog, plik elementarny):
- — `SC_FILE_TYPE_DF` – katalog
- — `SC_FILE_TYPE_WORKING_EF` – plik elementarny
- — `SC_FILE_TYPE_INTERNAL_EF` – plik elementarny

ef_structure wewnętrzna struktura pliku:
- — `SC_FILE_EF_TRANSPARENT` – plik o strukturze transparentnej
- — `SC_FILE_EF_LINEAR_FIXED` – plik o strukturze liniowej
- — `SC_FILE_EF_LINEAR_FIXED_TLV` – plik o strukturze liniowej o budowie TLV

— SC_FILE_EF_LINEAR_VARIABLE – plik o strukturze liniowej zmiennej
— SC_FILE_EF_LINEAR_VARIABLE_TLV – plik o strukturze liniowej zmiennej o budowie TLV
— SC_FILE_EF_CYCLIC – plik cykliczny
— SC_FILE_EF_CYCLIC_TLV – plik cykliczny o budowie TLV
— SC_FILE_EF_UNKNOWN – inny plik

size wielkość pliku (w bajtach)
id identyfikator pliku (2 bajty)
record_length, record_count dla plików o strukturze rekordowej określa wielkość rekordu oraz ich ilość

Wybranie już istniejącego obiektu z karty (komenda SELECT FILE) przy użyciu ścieżki do tego obiektu umożliwia funkcja:

```
int sc_select_file(sc_card_t       *card,
                   const sc_path_t *path,
                   sc_file_t       **result);
```

card kontekst połączenia z kartą
path ścieżka do obiektu
result struktura reprezentująca plik
wartość zwracana wartość ujemna w przypadku błędu

W przypadku tworzenia nowych obiektów w pierwszej kolejności konieczne jest utworzenie struktury reprezentującej ten obiekt, a następnie wypełnienie tej struktury odpowiednimi wartościami. Po wykorzystaniu tego obiektu należy zwolnić zajmowane przez niego zasoby. Umożliwiają to funkcje:

```
sc_file_t *sc_file_new(void);
void sc_file_free(sc_file_t *file);
```

file struktura reprezentująca element systemu plików
wartość zwracana w przypadku pierwszej z funkcji - struktura reprezentująca element systemu plików

Na fizyczne utworzenie elementu systemu plików w karcie lub jego usunięcie pozwalają funkcje:

```
int sc_create_file(struct sc_card       *card,
                   struct sc_file       *file);
int sc_delete_file(struct sc_card       *card,
                   const struct sc_path *path);
```

card kontekst połączenia z kartą
file struktura reprezentująca plik
path ścieżka do obiektu
wartość zwracana wartość ujemna w przypadku błędu

Operacje na plikach o strukturze przezroczystej, takie jak zapis, odczyt czy uaktualnienie zawartości (odpowiedniki komend systemu operacyjnego) są wykonywane z użyciem funkcji:

```
int sc_read_binary(sc_card_t     *card,
                   unsigned int   offset,
                   unsigned char *buffer,
                   size_t         count,
```

```
                  unsigned long flags);
int sc_write_binary(sc_card_t           *card,
                    unsigned int        offset,
                    const unsigned char *buffer,
                    size_t              count,
                    unsigned long       flags);
int sc_update_binary(sc_card_t          *card,
                     unsigned int       offset,
                     const unsigned char *buffer,
                     size_t             count,
                     unsigned long      flags);
```

card kontekst połączenia z kartą
offset położenie w pliku od którego należy rozpocząć operację
buffer bufor na dane (z danymi)
count wielkość bufora
flags parametr aktualnie nieużywany, powinien być ustawiony na 0
wartość zwracana wartość ujemna w przypadku błędu

Analogiczne funkcje zostały zaimplementowane dla plików o strukturze rekordowej. Umożliwiają one odczyt, zapis, uaktualnienie oraz dodanie rekordu w ramach danego pliku.

```
int sc_read_record(sc_card_t     *card,
                   unsigned int  rec_nr,
                   unsigned char *buffer,
                   size_t        count,
                   unsigned long flags);
int sc_write_record(sc_card_t           *card,
                    unsigned int        rec_nr,
                    const unsigned char *buffer,
                    size_t              count,
                    unsigned long       flags);
int sc_update_record(sc_card_t          *card,
                     unsigned int       rec_nr,
                     const unsigned char *buffer,
                     size_t             count,
                     unsigned long      flags);
int sc_append_record(sc_card_t          *card,
                     const unsigned char *buffer,
                     size_t             count,
                     unsigned long      flags);
```

card kontekst połączenia z kartą
rec_nr numer rekordu
buffer bufor na dane (z danymi)
count wielkość bufora
flags parametr aktualnie nieużywany, powinien być ustawiony na 0
wartość zwracana wartość ujemna w przypadku błędu

Przykładem funkcji, która umożliwia operowanie na strukturze PKCS #15 jest metoda pozwalająca na wykonanie podpisu cyfrowego z użyciem algorytmu RSA. Dane przekazywane są z użyciem bufora wejściowego, a sam podpis, w odpowiednim formacie, zostanie umieszczony w buforze wyjściowym. Pierwsze trzy parametry funkcji identyfikują kartę oraz obiekt z użyciem którego należy wykonać operację.

```
int sc_pkcs15_compute_signature(struct sc_pkcs15_card         *p15card,
```

```
                                const struct sc_pkcs15_object *obj,
                                unsigned long                 flags,
                                const u8                      *in,
                                size_t                        inlen,
                                u8                            *out,
                                size_t                        outlen);
```

Uwagi bibliograficzne

Z zaleceniami PKCS #15 i #11 można zapoznać się w [65,66,64,67]. Liczne przykłady wykorzystania interfejsu PKCS #11 zamieszczono w [85].

OpenSC jest opisane w [82].

11. System GSM

System GSM (ang. *Global System for Mobile Communications*) jest systemem telefonii komórkowej. Głównym tematem tego rozdziału jest karta SIM (ang. *subscriber identity module*), która jest kartą procesorową identyfikującą abonenta w sieci telefonicznej.

11.1. Działanie systemu

Zgodnie z dokumentem GSM 01.02, który opisuje podstawy sieci GSM, można wyróżnić w niej trzy podstawowe podsystemy:

— RSS, podsystem radiowy (ang. *radio subsystem*) – złożony z telefonu komórkowego (ME, ang. *mobile equipment*) wraz z kartą SIM tworzącego stację ruchomą (MS, ang. *mobile station*) oraz zespół stacji bazowych (BSS, ang. *base station subsystem*) złożonych z jednej lub kilku stacji bazowych (BST, ang. *base transceiver station*) oraz kontrolera (sterownika) stacji bazowych (BSC, ang. *base station controller*)

— NSS, podsystem komutacyjno-sieciowy (ang. *network and switching subsystem*) – zawiera centralę systemu ruchomego (MSC, ang. *mobile switching center*) oraz rejestr stacji obcych (VLR, ang. *visitor location register*)

— OMS, zespół eksploatacji i utrzymania (ang. *operation and maintenance subsystem*) – wyróżnia się w nim centra eksploatacji i utrzymania (OMC, ang. *operation and maintenance center*), centrum identyfikacji (AUC, ang. *authentication center*), rejestr identyfikacji wyposażenia (EIR, ang. *equipment identity register*) oraz rejestr stacji własnych (HLR, ang. *home location register*)

Z elementami podsystemu OMS oraz NSS związane są bazy danych przechowujące odpowiednie informacje o sieci, jej elementach i ich konfiguracji.

HLR zawiera informacje dotyczące abonenta. W danych tych zawarte są między innymi numery: IMSI (ang. *international mobile subscriber identity*, unikalny numer karty SIM w sieci) oraz MSISDN (ang. *mobile station ISDN number*, niezależny od IMSI numer telefoniczny abonenta). Baza HLR przechowuje także informacje o dostępie do usług dodatkowych jakie wybrał użytkownik (np. czy posiada dostęp do usługi GPRS), dane związane z uwierzytelnieniem abonenta w sieci oraz (jeśli są dostępne) informacje o aktualnym rejestrze VLR w jakim jest obecny użytkownik, jego numerze TMSI (ang. *temporary mobile subscriber identity*, tymczasowy numer abonenta ruchomego), MSRN (ang. *mobile station roaming number*, adres chwilowy stacji ruchomej) oraz lokalizacji MSC.

VLR przechowuje informacje dotyczące użytkowników tymczasowo przebywających na pewnym obszarze. Oprócz podstawowych informacji jakie gromadzi baza HLR znajdują się tu dodatkowo TMSI, MSRN oraz LAI (ang. *location area information*, numer obszaru przywołań). Informacje w bazie VLR zmieniają się dynamicznie (w zależności od położenia użytkownika).

EIR zawiera trzy listy numerów IMEI (ang. *international mobile equipment identity*), czyli międzynarodowych identyfikatorów terminali ruchomych: białą (wszystkie urządzenia jakie mogą być używane w sieci), szarą (włączenie się do sieci tych urządzeń powinno być raportowane) oraz czarną (urządzenia zastrzeżone).

Podczas uwierzytelnienia karty SIM w sieci stosowany jest algorytm A3, który jest zależny od operatora sieci komórkowej. Podobnie algorytm A8, którego wynikiem jest ustalenie klucza służącego do szyfrowania danych przesyłanych między urządzeniem ruchomym a stacją bazową. Procedura ta przebiega następująco:

— po włączeniu aparatu do systemu, złożonego z BSS i MSC, przesyłana jest ostatnia wartość LAI oraz TMSI albo (jeśli te dane nie są dostępne) IMSI

— system poddaje te informacje pewnemu przekształceniu otrzymując klucz Ki

— do aparatu przesyłana jest liczba losowa i na jej podstawie (algorytm A3 z kluczem Ki) obliczany jest SRES (ang. *signed response*, „podpis elektroniczny" wykorzystywany przy uwierzytelnianiu abonenta)
— podobną operację wykonuje system i porównuje z wynikiem otrzymanym od aparatu; jeśli wyniki są identyczne SIM jest uwierzytelniony w systemie
— na podstawie wcześniejszej liczby losowej i klucza Ki obliczany jest klucz Kc (algorytm A8) służący do szyfrowania danych przesyłanych pomiędzy aparatem a stacjami

11.2. Stuktura karty

Karta SIM w większości przypadków jest formatu ID-000 (zobacz 2.1.1).

Układ plików i katalogów w karcie SIM ma typową dla kart procesorowych hierarchiczną strukturę. W normie GSM 11.11 zdefiniowano przeszło 70 plików, z czego 12 jest obowiązkowych dla systemu (około 110 bajtów danych). Ilość plików zależna jest od operatora wydającego kartę. Przeciętna karta zawiera jednak około 40 plików o łącznej objętości około 12 kB.

Dzięki krótkim wiadomościom tekstowym możliwe jest zdalne zarządzanie zawartością plików oraz ich układem w karcie przez operatora systemu. Skrypty zawierające zestaw komend są automatycznie przetwarzane przez telefon.

W katalogu głównym (MF, FID = '3F00') znajdują się następujące obiekty:
— DF_{GSM} (FID = '7F20') – katalog zawiera obiekty specyficzne dla danej sieci telefonii GSM
— $DF_{Telecom}$ (FID = '7F10') – katalog zawiera obiekty specyficzne dla usług telefonii komórkowej
— EF_{ICCID} (FID = '2FE2', ang. *integrated circut card identyfication*) – unikalny numer karty procesorowej
— EF_{ELP} (FID = '2F05', ang. *extended language preference*) – lista języków dla interfejsu użytkownika telefonu (najbardziej oczekiwany jest pierwszy na liście)

Zawartość katalogu DF_{GSM} jest swoistą konfiguracją sieci do jakiej należy karta:
— EF_{ACM} (FID = '6F39', ang. *accumulated call meter*) – w pliku tym przechowywana jest długość wykonanych rozmów (impulsy) od pewnego punktu w przeszłości; plik ma strukturę cykliczną
— EF_{ACMmax} (FID = '6F37', ang. *accumulated call meter maximum*) – maksymalna długość (impulsy) wykonanej rozmowy
— EF_{FPLMN} (FID = '6F7B', ang. *forbidden public land mobile network*) – plik ten zawiera listę zabronionych operatorów telefonii komórkowej
— EF_{HPLMN} (FID = '6F31', ang. *home public land mobile network search period*) – znajduje się tu informacja o okresie czasu jaki upływa pomiędzy operacją wyszukiwania domowej sieci
— EF_{IMSI} (FID = '6F07', ang. *international mobile subscriber identity*) – w pliku przechowywany jest międzynarodowy numer abonenta sieci komórkowej składający się z identyfikatora operatora oraz numeru seryjnego
— $EF_{PLMNsel}$ (FID = '6F30', ang. *public land mobile network selector*) – plik zawiera listę preferowanych operatorów telefonii komórkowej
— EF_{LOCI} (FID = '6F7E', ang. *location information*) – plik zawiera aktualne informacje dotyczące lokalizacji abonenta np. tymczasowy numer identyfikacyjny abonenta (TMSI, ang. *temporary mobile subscriber identity*) oraz informacje o lokalizacji (LAI, ang. *location area information*); ze względu na częste zmiany tych informacji często aparat telefoniczny tymczasowo przechowuje odpowiednie dane zapisując je tylko w ostateczności
— EF_{LP} (FID = '6F05', ang. *language preference*) – lista języków interfejsu użytkownika preferowanych przez właściciela karty

— EF_{PHASE} (FID = ’6FAE’, ang. *phase information*) – plik zawiera informacje o wersji (fazie rozwoju) systemu GSM wspieranego przez kartę
— EF_{PUCT} (FID = ’6F41’, ang. *price per unit and currency table*) – plik zawiera informacje o walucie i cenie za jedną jednostkę połączenia
— EF_{SPN} (FID = ’6F46’, ang. *service provider name*) – nazwa operatora sieci komórkowej
— EF_{SST} (FID = ’6F38’, ang. *SIM service table*) – plik zawiera informację o dostępnych i aktywowanych usługach poza rozmowami telefonicznymi takimi jak SMS lub GPRS
— EF_{KC} (FID = ’6F20’, ang. *Kc key*) – klucz kryptograficzny przeznaczony do szyfrowania danych przesyłanych do BTS
— $DF_{GRAPHICS}$ (FID = ’5F50’, ang.) – katalog zawiera pliki z obrazami wyświetlanymi na ekranie telefonu (np. logo operatora)
  — EF_{IMG} (FID = ’4F20’, ang.) – plik ten zawiera wskaźniki do innych plików w tym katalogu, które zawierają już właściwe rysunki

Dane związane z usługami telekomunikacyjnymi przechowywane są w katalogu $DF_{Telecom}$. Do najważniejszych plików zaliczyć można:
— EF_{ADN} (FID = ’6F3A’, ang. *abbreviated dialing numbers*) – plik zawiera listę numerów telefonicznych (książka telefoniczna)
— EF_{FDN} (FID = ’6F3B’, ang. *fixed dialing numbers*) – lista numerów telefonicznych na które można wykonywać połączenia z aparatu (połączenia na numery spoza tej listy będą zabronione)
— EF_{LND} (FID = ’6F44’, ang. *last number dialed*) – lista ostatnio używanych numerów telefonicznych
— EF_{MSISDN} (FID = ’6F40’, ang. *mobile station ISDN number*) – plik zawiera numer aparatu telefonicznego
— EF_{SDN} (FID = ’6F49’, ang. *service dialing numbers*) – lista numerów do usług operatora sieci komórkowej
— EF_{SMS} (FID = ’6F3C’, ang. *short message service*) – w pliku tym przechowywane są krótkie wiadomości tekstowe wysłane i otrzymane z sieci; plik ma strukturę rekordową (każdy z rekordów o wielkości 176 bajtów)
— EF_{SMSP} (FID = ’6F42’, ang. *short message service parameters*) – plik zawiera ustawienia związane z wysyłaniem krótkich wiadomości tekstowych
— EF_{SMSS} (FID = ’6F43’, ang. *short message service status*) – plik zawiera informacje o statusie zachowanych w karcie SIM krótkich wiadomości tekstowych

Początkowo zakładano, że karta SIM będzie wymieniana co około dwa lata. W rzeczywistości jednak żywotność tej karty (podstawowym problemem jest ograniczona ilość zapisów do pamięci) jest kilkakrotnie dłuższa. Aparaty telefoniczne konstruowane są w taki sposób by w ostateczności zapisywać informacje w karcie, co zmniejsza ilość wykonywanych operacji.

Funkcjonalność kart wykorzystywanych w telefonii komórkowej wciąż wzrasta. Telefon nie służy już tylko do rozmów, ale również pełni rolę np. elektronicznej portmonetki lub narzędzia umożliwiającego uwierzytelnienie. Coraz częściej wykorzystywane są karty z maszyną wirtualną języka Java. W połączeniu z midletami otrzymać można spójne i bezpieczne aplikacje w aparacie telefonicznym. Możliwość przejęcia kontroli nad aparatem przez kartę (specjalna komenda FETCH, której odpowiedzi aparat traktuje jak rozkazy) otwiera nowe, zupełnie nowe możliwości tworzenia aplikacji kartowych.

Rozkazy obsługiwane przez telefon opisano w dokumencie GSM 11.14.

11.3. Narzędzia

Istnieje bardzo wiele programów pozwalających operować na karcie SIM. Główne możliwości takich narzędzi to odczyt i zapis danych znajdujących się w karcie (książka telefoniczna, krótkie wiadomości tekstowe). Informacje o wielu komercyjnych produktach tego typu można znaleźć w Internecie. Innego typu aplikacje pozwalają na obserwację zachowania i parametrów sieci GSM. Jednym z nich jest *Net monitor*, w który są wyposażone niektóre z aparatów telefonicznych.

Poniżej przedstawione zostanie proste, powszechnie dostępne narzędzie *gsmcard* oraz podstawowe możliwości *Net monitor-a*.

11.3.1. gsmcard

Program *gsmcard*, pisany z myślą o wykorzystaniu w środowisku *Linux*, dostępny jest pod adresem `http://www.linuxnet.com`. Po uruchomieniu skryptu w języku Perl, będącym konsolową nakładką dla programu w C, możliwe jest zastosowanie następujących poleceń:

— *help* – podręczna pomoc do opcji programu; możliwe jest uzyskanie szczegółowej pomocy do danej komendy po podaniu jej nazwy
— *opencard* – otwarcie sesji z kartą, opcjonalnie można podać kod PIN
— *forget* – usunięcie zapamiętanych kodów PIN pamięci komputera
— *chgpin* – zmiana kodu PIN (pierwszego lub drugiego)
— *givepin* – weryfikacja kodu PIN (pierwszego lub drugiego)
— *reqpin1* – włączenie lub wyłączenie żądania kodu PIN
— *unblockpin* – odblokowanie kodu PIN (pierwszego lub drugiego) z użyciem kodu PUK
— *status* – wydruk informacji o aktualnym stanie połączenia z kartą
— *info* – pobranie z odpowiedniego pliku z karty informacji o: numerze seryjnym (opcja *serial*), preferowanym języku (opcja *lang*), numerze IMSI (opcja *imsi*), operatorze (opcja *sp*)
— *write* – zapis książki telefonicznej z lokalnego pliku (plik w każdej linii zawiera numer porządkowy, opis oraz numer telefoniczny)
— *read* – odczyt do pliku lokalnego książki telefonicznej z karty z możliwością pominięcia pustych rekordów
— *closecard* – zamknięcie sesji z kartą
— *quit* – wyjście z programu

Operacje odczytu z karty nie powinny doprowadzić do nieprzewidzianych skutków, natomiast należy być szczególnie ostrożnym przy zapisie informacji.

11.3.2. Net monitor

Narzędzie to pozwala obserwować i zmieniać parametry telefonu oraz sieci komórkowej. Poszczególne ekrany dostępne są bezpośrednio w aparacie (po aktywacji narzędzia). Nie wszystkie telefony wyposażone są w *Net monitor-a* lub podobne narzędzie. Numeracja ekranów w tym rozdziale zgodna jest z aplikacją w telefonach firmy *Nokia*.

Najciekawsze informacje jakie można odczytać dotyczą między innymi:

— ekrany 1 i 2 – numer aktualnej stacji, która obsługuje telefon oraz parametry kanału na jakim jest połączenie (prędkość transmisji itp.)
— ekrany 3, 4 i 5 – informacje o stacjach jakie są w zasięgu aparatu
— ekran 6 – po lewej stronie wyświetlony jest identyfikator sieci domowej i sieci dozwolonych, po prawej identyfikatory sieci zastrzeżonych
— ekran 7 – szereg parametrów sieci np. czy dozwolone są telefony alarmowe
— ekran 10 – m. in. aktualna wartość parametru TMSI
— ekran 12 – m. in. parametry dotyczące szyfrowania

— ekran 20 – parametry określające stan ładowania baterii telefonu (np. napięcie na wyjściu baterii i jej temperatura); dodatkowe informacje dotyczące baterii są na kolejnych trzech ekranach
— ekran 35 – stan karty SIM po ostatnim zerowaniu
— ekran 51 – informacje o karcie SIM takie jak: napięcie zasilania, ilość pozostałych prób dla kodów PIN oraz PUK, prędkość transmisji danych

Net monitor najczęściej wykorzystywany jest przez serwisantów sieci komórkowych do badania aktualnych jej parametrów. Pozwala również na zdiagnozowanie uszkodzenia aparatu lub stacji w sieci.

Uwagi bibliograficzne

Najważniejszymi dokumentami opisującymi standard telefonii komórkowej są [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63].

12. Cykl życia karty

W rozdziale tym przedstawiony jest cykl życia karty (a tym samym aplikacji kartowej). Opisano również specyfikację **GlobalPlatform**, która jest próbą uporządkowania tematyki zarządzania systemami kartowymi.

12.1. ISO 10202-1

Norma ISO 10202-1 definiuje pięć etapów w cyklu życia karty:
— produkcja procesora oraz karty,
— przygotowanie systemu operacyjnego karty,
— personalizacja,
— użytkowanie karty,
— zakończenie użytkowania karty.
Każdy z tych etapów został opisany dokładniej w poniższych podrozdziałach.

12.1.1. Produkcja procesora oraz karty

Pierwszym krokiem zmierzającym do wyprodukowania karty procesorowej jest projekt samego układu elektronicznego. W początkowym stadium jest to zazwyczaj model procesora (symulator), który poddawany jest licznym testom logicznym. Następnie wytwarza się rzeczywisty układ. Stworzenie nowego procesora wiąże się zazwyczaj ze stworzeniem całej technologii jego produkcji i trwa to zazwyczaj od roku do trzech lat.

System operacyjny karty (lub jego podstawowe elementy) umieszczane są w masce ROM. Implementacja odbywa się przy użyciu układu, w którym pamięć ROM zastąpiono pamięcią EEPROM. Testowanie maski ROM jest bardzo istotne ze względu na duże koszty jej produkcji.

Układy produkowane są w postaci płyt krzemowych zawierających kilkaset procesorów. Po przetestowaniu poprawności działania płyta jest cięta na poszczególne kości, które następnie umieszczane są w modułach. Zewnętrzną częścią modułu są styki, które są interfejsem do czytnika kart. Układ, po umieszczeniu w module, jest ponownie testowany. Na tym etapie sprawdzane jest czy moduł zwraca poprawny ATR. Wywoływane są również komendy systemu przeprowadzające wewnętrzne testy procesora, pamięci i układów pomocniczych. Moduły zazwyczaj (przed wtopieniem w plastik karty) przechowywane są w postaci taśm.

Po wyprodukowaniu i wstępnym nadruku plastikowych kart moduły są w nie wtapiane (wklejane). Zaraz po tej operacji przeprowadzane są ponowne testy działania karty. Na tym etapie w karcie mogą zostać zapisane następujące informacje:
— identyfikator producenta układu elektronicznego,
— identyfikator określający fabrykę, w którym wyprodukowano układ,
— unikalny numer układu elektronicznego,
— rodzaj (wersja) układu elektronicznego,
— identyfikator modułu, w który wtopiony będzie układ,
— data oraz czas określający moment wtopienia układu w kartę.

Na tym kończy się mechaniczny proces produkcji karty.

12.1.2. Przygotowanie systemu operacyjnego

W początkowej fazie tego etapu przeprowadzane są testy obejmujące elektryczne właściwości karty (testy pamięci i układów logicznych) oraz poprawność transmisji danych. Po ich zakończeniu wykonuje się operację kompletowania systemu operacyjnego karty. Aby uprościć wstępny etap produkcji oraz zminimalizować koszty ewentualnych błędów w masce ROM właściwą część systemu operacyjnego dodaje się na tym etapie życia karty. Funkcje karty, które

pozwoliły na uzupełnienie systemu operacyjnego oraz testowanie jej wewnętrznych elementów zostają bezpowrotnie wyłączone.

Inicjalizacja karty polega na założeniu w systemie plików niezbędnych katalogów i plików oraz umieszczeniu kluczy transportowych w karcie. Na tym etapie w karcie mogą być zapisane dodatkowo następujące informacje:
— identyfikator osoby (instytucji) przeprowadzającej opisane operacje,
— identyfikator maszyny,
— data i czas operacji.
Dane te nie mają związku z przyszłym właścicielem karty oraz sposobem jej wykorzystania.

12.1.3. Personalizacja

Wyróżniamy personalizację wizualną oraz elektryczną.

Personalizacja wizualna polega na nadruku lub wytłoczeniu na karcie specyficznych informacji np. danych właściciela i jego zdjęcia.

Personalizacja elektryczna związana jest z umieszczeniem aplikacji w karcie. W zależności od typu aplikacji może to być związane z umieszczeniem kluczy kryptograficznych w karcie oraz dodatkowo wydruku kodów PIN dla przyszłego właściciela.

Obie części personalizacji dokonywane są zazwyczaj równolegle przy wykorzystaniu specjalnych maszyn.

Na tym etapie w karcie mogą być zapisane następujące informacje:
— identyfikator osoby (instytucji) umieszczającej aplikację w karcie,
— identyfikator maszyny,
— data i czas personalizacji.

Personalizacja jest procesem, który powinien przebiegać w bezpiecznym środowisku.

Etap ten kończy się z momentem przekazania działającej karty osobie, która ma z niej korzystać w życiu codziennym.

12.1.4. Użytkowanie karty

Podczas użytkowania karty mogą być umieszczane w niej nowe aplikacje, a już istniejące mogą być czasowo blokowane lub całkowicie wyłączane z użycia. Jednakże głównie etap ten polega na korzystaniu z funkcjonalności aplikacji w karcie.

12.1.5. Zakończenie użytkowania karty

Zakończenie procesu użytkowania karty związane jest z nieodwracalnym zablokowaniem wszystkich aplikacji jakie wcześniej w niej umieszczono. Dane nadrukowane na karcie również powinny stać się nieczytelne.

Najczęściej stosowanym sposobem jest fizyczne zniszczenie karty (pocięcie jej na kawałki).

12.2. GlobalPlatform

Specyfikacja ta, początkowo rozwijana przez *Visa*, nosiła nazwę *Open Platform*. Obecnie organizacja skupiająca twórców tego standardu występuje pod godłem *GlobalPlatform*. Jej celem jest stworzenie infrastruktury umożliwiającej współpracę wielu dostawców sprzętu i aplikacji w jednym systemie kartowym.

Oprócz wytycznych dla kart (*GlobalPlatform Card Specification*) organizacja pracuje również nad zaleceniami dla urządzeń korzystających z kart, takich jak czytniki, terminale, bankomaty (*GlobalPlatform Device Specification*) oraz dla elementów systemu wykorzystujących karty, na jakie składają się np. systemy zarządzania kluczami, system wymiany komunikatów, systemy zarządzania kartami (*GlobalPlatform Systems Specification*).

Do podstawowych komponentów, z jakich złożona jest karta zgodna z *GlobalPlatform* należą:

— środowisko uruchomieniowe złożone z systemu operacyjnego karty, maszyny wirtualnej oraz API umożliwiającego dostęp do systemu operacyjnego z poziomu aplikacji wykonywanych przez maszynę wirtualną,
— API *GlobalPlatform*, które jest zestawem komend pozwalającym na zarządzanie kartami wieloaplikacyjnymi podczas ich eksploatacji oraz na korzystanie z kluczy kryptograficznych do zabezpieczenia przesyłanych informacji w ramach aplikacji od jednego dostawcy,
— zarządca karty (ang. *Card Manager*), który odpowiedzialny jest za: przekazywanie komend do określonych aplikacji, zarządzanie zawartością (aplikacjami) w karcie, zarządzanie bezpieczeństwem aplikacji (przechowywanie współdzielonych kodów PIN oraz kluczy kryptograficznych), zarządzanie domenami bezpieczeństwa (ang. security domains),
— domeny bezpieczeństwa, które umożliwiają logiczne rozdzielenie zasobów współdzielonych przez określone aplikacje,
— aplikacje kartowe o różnej funkcjonalności.

GlobalPlatform wyróżnia następujące etapy w cyklu życia karty (reprezentowanej przez zarządcę karty):

— OP_READY – gotowa,
— INITIALIZED – zainicjalizowana (przejście z poprzedniego stanu poprzez załadowanie kluczy do karty),
— SECURED – zabezpieczona (oznacza to, że karcie dostarczono wszystkie niezbędne klucze i informacje służące do zabezpieczonej komunikacji),
— CARD_LOCKED – zablokowana,
— TERMINATED – permanentnie zablokowana.

Ponadto zostały określone etapy w cyklu życia danej aplikacji:

— LOADED – aplikacja jest umieszczona w karcie,
— INSTALLED – aplikacja jest zainstalowana w karcie,
— SELECTABLE – aplikacja jest gotowa do użycia,
— stany związane ze specyfiką danej aplikacji,
— LOCKED – aplikacja jest zablokowana.

Specyfikacja *GlobalPlatform* szczegółowo określa jakie komendy i w jaki sposób powinny być zaimplementowane w karcie. Należą do nich:

— DELETE – usunięcie instancji aplikacji, aplikacji lub klucza z karty,
— GET DATA – pobranie danych o określonych obiektach z karty (np. numer identyfikacyjny wydawcy, informacje o kluczach),
— GET STATUS – umożliwia odczytanie aktualnego stanu obiektów takich jak aplikacja czy plik wykonywalny (zwracane są np. informacje o AID aplikacji i jej przywilejach),
— INSTALL – przeznaczona do zarządzania zawartością karty poprzez np. tworzenie instancji apletów w karcie,
— LOAD – wykorzystywana do przekazywania bloków danych do karty,
— MANAGE CHANNEL – przeznaczona do zarządzania komunikacji z wykorzystaniem kanałów logicznych,
— PUT KEY – pozwala na umieszczenie w karcie nowego klucza (kluczy) lub podmianę starego klucza (kluczy) na jego nową wersję,
— SELECT – pozwala wybrać aktualną aplikację z karty poprzez jej AID,
— SET STATUS – przeznaczona do zarządzania cyklem życia karty lub aplikacji,
— STORE DATA – przeznaczona do przekazywania danych do aplikacji lub domeny bezpieczeństwa,

— INITIALIZE UPDATE – inicjalizacja kanału do zabezpieczonego przesyłania danych,
— EXTERNAL AUTHENTICATE – uwierzytelnienie hosta oraz pobranie informacji o wymaganym przez niego poziomie bezpieczeństwa,
— BEGIN R-MAC SESSION oraz END R-MAC SESSION – pozwalają na rozpoczęcie i zakończenie sesji komunikacyjnej, w której sprawdzana jest integralność przesyłanych danych.

12.2.1. Interfejs programisty

Niezalecana obecnie, choć wykorzystywana w produktach znajdujących już się na rynku wersja API składa się z:

openplatform.OPSystem klasa ta umożliwia dostęp do publicznych metod zarządcy karty pozwalających między innymi na: ustawienie oraz odczytanie aktualnego stanu karty w cyklu życia, zablokowanie karty, ustawienie oraz weryfikację kodu PIN; wszystkie metody tej klasy są metodami statycznymi,

openplatform.ProviderSecurityDomain implementacja tego interfejsu, który jest obiektem współdzielonym, dostarcza funkcjonalności zabezpieczonej komunikacji (metody otwierające i zamykające bezpieczny kanał), możliwości uwierzytelnienia oraz zarządzania kluczami.

Aktualna wersja pakietu (zalecana), oprócz zmiany nazwy, charakteryzuje się rozdzieloną funkcjonalnością głównej klasy na dwa dodatkowe interfejsy:

org.globalplatform.GPSystem metody tej klasy pozwalają na zarządzanie aktualnym statusem karty - jej chwilowym lub całkowitym zablokowaniem; ponadto pozwalają na dostęp do uchwytów do obiektów reprezentujących zaimplementowane interfejsy reprezentujące kod PIN oraz kanał zabezpieczonej komunikacji,

org.globalplatform.Application interfejs ten musi być zaimplementowany aby umożliwić apletowi dodatkową funkcjonalność pozwalającą na przetwarzanie danych otrzymanych z innego obiektu w danej domenie bezpieczeństwa,

org.globalplatform.SecureChannel metody tego interfejsu przeznaczone są do zarządzania zabezpieczonymi kanałami przesyłu danych; pozwalają one na implementację określonego sposobu bezpiecznej komunikacji,

org.globalplatform.CVM interfejs ten reprezentuje szeroko pojęty kod PIN (sposób uwierzytelnienia właściciela karty); pozwala na określenie sposobu jego zapisu oraz zarządzanie nim.

Uwagi bibliograficzne

Cykl życia karty precyzuje norma [4].

Specyfikacja *GlobalPlatform* szczegółowo opisana jest w [37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49].

13. Ataki na systemy wykorzystujące karty elektroniczne

W rozdziale przedstawione są różne ataki na karty inteligentne oraz na systemy z nimi związane. Opisano również sposoby ochrony przed tymi atakami.

13.1. Elementy bezpieczeństwa systemów kartowych

Na bezpieczeństwo systemu kartowego składa się bezpieczeństwo jego poszczególnych komponentów do których zaliczamy:
— część plastikową karty oraz nadruk,
— układ elektroniczny,
— system operacyjny karty,
— aplikację kartową,
— system zewnętrzny,
— urządzenia w których wykorzystywana jest karta (terminale, bankomaty).
Ochrona polega na odpowiednim zabezpieczeniu wymienionych elementów przed nieuprawnionym dostępem mającym na celu modyfikację lub wyłącznie poznanie struktury danego modułu.

13.2. Klasyfikacja ataków i atakujących

Ze względu na warstwę systemu w jakiej dokonywany jest atak możemy wyróżnić:
— atak społeczny – mający na celu np. przekupienie osoby projektującej system lub mającej udział w jego implementacji w celu wydobycia poufnych danych; atak ten może też dotyczyć właścicieli kart, którzy mogą np. w sprytny sposób zostać oszukani,
— atak fizyczny – polega na fizycznym dostępie do karty elektronicznej z użyciem specjalistycznego sprzętu; dodatkowo możemy wyróżnić:
  — atak statyczny – jest to analiza budowy karty bez dostarczenia jej zasilania,
  — atak dynamiczny – polega na analizie właściwości karty podczas jej funkcjonowania;
— atak logiczny – wiąże się z kryptoanalizą, modyfikacją systemu zewnętrznego (poprzez wykorzystanie np. koni trojańskich) lub wykorzystaniu błędów w systemie operacyjnym karty; w tej warstwie możemy wyróżnić ataki:
  — aktywne – atakujący modyfikuje dane przekazywane w systemie (np. zmusza kartę do obliczania specyficznych kryptogramów),
  — pasywne – atakujący jedynie nasłuchuje i przechwytuje dane wymieniane w systemie.

Ataki mogą wiązać się z cyklem życia karty (zobacz 12). W takim przypadku można dokonać następującej klasyfikacji:
— ataki podczas projektowania układu karty i systemu operacyjnego,
— ataki podczas produkcji układu i karty,
— ataki podczas użytkowania karty w systemie.

Spośród osób (organizacji) atakujących systemy kartowe możemy wyróżnić: hakerów, naukowców oraz kryminalistów. Podział ten związany jest z celem ataku. Może on mieć na celu poprawienie bezpieczeństwa systemu, jego kompromitację lub próbę np. uzyskania nielegalnych środków finansowych.

13.3. Ataki oraz sposoby ochrony

Podczas projektowania układu elektronicznego oraz systemu operacyjnego karty mogą być popełnione błędy, które umożliwią przyszły atak. Ważna jest metodyka projektowania - wiedza o tworzonym produkcie nie powinna być w rękach jednego człowieka. Ponadto niebagatelnym

problemem jest odpowiednie zabezpieczenie środowiska pracy programistów. Informacje o budowie układu oraz systemie operacyjnym mogą uprościć późniejszą jego analizę. Jednak w profesjonalnie zaprojektowanym systemie kartowym nawet analiza kodów źródłowych nie powinna pomóc w nieuprawnionym dostępie do danych w karcie.

Oznaczenie każdego wyprodukowanego układu unikalnym numerem umożliwi blokadę kart na których stwierdzono próby modyfikacji oraz ułatwi zarządzanie kartami w systemie.

Na etapie produkcji kart kradzież układów z niekompletnym systemem operacyjnym może ułatwić późniejszy atak lub wydobycie kluczy transportowych z karty.

Sposobami obrony przed statyczną analizą układu elektronicznego karty są:
— technologia produkcji oraz zastosowanie materiałów wysokiej jakości,
— stosowanie w układzie nieaktywnych struktur logicznych (w celu skomplikowania analizy układu),
— szyny łączące procesor z pamięcią powinny być wbudowane wewnątrz układu,
— pamięć ROM w celu ochrony przed analizą optyczną pod mikroskopem powinna być zanurzona w warstwie silikonu,
— zapis do pamięci nie powinien być liniowy w celu utrudnienia analizy odczytanych danych,
— zabezpieczenie pamięci RAM i EEPROM przed nieuprawnionym odczytem poprzez ich odpowiednie umiejscowienie w układzie,
— zniszczenie układu przy próbie modyfikacji elementów elektronicznych na drodze reakcji chemicznych.

Dynamiczna analiza układu polega na obserwowaniu zachowania karty podczas jej działania w różnych warunkach (modyfikacja częstotliwości pracy układu, zasilania itp.). Ochrona przed różnymi typami tych ataków polega między innymi na:
— monitoring napięcia na stykach karty – karta nie powinna pracować w zbyt niskim lub wysokim napięciu; nieprawidłowe napięcie może doprowadzić do błędnego wykonywania kodu w karcie,
— monitoring częstotliwości taktowania układu – nieprawidłowa częstotliwość może doprowadzić do niestabilności pracy układu a tym samym do możliwości wycieku z karty chronionych danych,
— monitoring temperatury pracy układu,
— nieodwracalna blokada funkcji systemu operacyjnego służących do ładowania dodatkowych jego komponentów i testowania układu,
— nieliniowy zapis danych do pamięci RAM,
— odpowiedniej budowie układu zabezpieczającej przed:
  — prostą analizą poboru mocy (ang. *simple power analysis*, SPA),
  — różnicową analizą poboru mocy (ang. *different power analysis*, DPA);
    Na podstawie analizy pobieranej mocy przez układ przy wykonywaniu np. operacji kryptograficznych można wywnioskować parametry wykonania tych algorytmów.
— zabezpieczenie przed ulotem elektromagnetycznym układu.

Podczas użytkowania karty możemy wyróżnić następujące rodzaje ataków w warstwie logicznej:
— analiza komunikacji z kartą, stworzenie podróbki karty lub ingenercja w proces komunikacji,
— nagłe odcięcie zasilania np. w momencie wykonywania operacji zapisu do pamięci (w źle zaprojektowanych układach może nastąpić przekłamanie w wartości danej zmiennej),
— analiza pracy karty przy nieprawidłowych parametrach (ang. *differential fault analysis*, DFA);
  Metoda ta polega na analizie wyników działania karty dla nieprawidłowych wartości np. parametrów algorytmów kryptograficznych.

— analiza czasowa np. wykonania operacji kryptograficznych;
  W zależności od parametrów algorytmów kryptograficznych wykonują się one w różnym czasie co pozwala, przy odpowiedniej bazie wcześniejszych pomiarów, na ich złamanie. Sposobem ochrony jest stosowanie implementacji, których czas wykonania nie jest zależny od parametrów.
— analiza działania generatora liczb pseudolosowych;
  Złamanie algorytmu generowania liczb pseudolosowych lub uszkodzenie generatora może pozwolić na możliwość modyfikacji danych w karcie. Ochrona polega na odpowiedniej implementacji generatora.
— analiza działania systemu operacyjnego – obejmuje analizę zachowania karty podczas operacji np. zapisu do pamięci, zachowania systemu podczas dostępu do danych przy różnych uprawnieniach.
  Ochrona polega na implementacji systemu operacyjnego w taki sposób, aby utrudnić analizę jego zachowania. Ponadto system operacyjny powinien być wolny od błędów. Szczególnie dotyczy to operacji, które bazują na analizie uprawnień użytkownika.

Do ataków w warstwie logicznej zaliczamy również ataki związane z innymi elementami systemu kartowego takimi jak: terminal, aplikacja działająca w terminalu, system obsługi kart itp. Ze względu na źródło oraz cel takich ataków można wyróżnić następujące ich rodzaje:

— atak z poziomu terminala na dane zawarte w karcie;
  Użytkownik karty używając jej np. w bankomacie lub terminalu płatniczym musi mieć zaufanie do tego urządzenia oraz aplikacji, która obsługuje jego kartę. Atak może polegać na zmianie parametrów transakcji np. użytkownik płaci za usługę 10 PLN a terminal automatycznie dolicza do kwoty 5 PLN. Dodatkowo terminal może zachowywać w pamięci dane o karcie (np. nazwisko właściciela) mimo iż potrzebne one są jedynie w czasie transakcji.
  Sposobami ochrony przed tymi atakami jest ciągła kontrola parametrów aplikacji kartowej i szybka reakcja u wydawcy karty po zauważeniu anormalnego zachowania. Dodatkowo właściciel terminala oraz wydawca karty mogą stale monitorować system. Ponadto należy korzystać jedynie z zaufanych urządzeń. Wykorzystywane aplikacje powinny być poddane audytowi, a urządzenia płatnicze odpowiednio zabezpieczone przed nieuprawnionymi zmianami w aplikacji. W przypadku aplikacji wykonywanych na komputerach osobistych należy korzystać z systemów co do których mamy pewność, że nie mają np. zainstalowanych koni trojańskich.
— atak właściciela karty przeciw aplikacji terminala;
  Atak ten polega na zastosowaniu podrobionej lub zmodyfikowanej karty w terminalu.
  Ochrona polega na uniemożliwieniu użytkownikowi samodzielnych modyfikacji w aplikacji kartowej oraz na każdorazowym uwierzytelnieniu karty przy użyciu w terminalu.
— atak posiadacza karty na dane w niej zawarte;
  Atak polega na próbie odczytania (np. kodów PIN, kluczy kryptograficznych) lub modyfikacji danych w karcie przez osoby nieuprawnione.
  Obrona z jednej strony powinna polegać na prawidłowym logicznym zabezpieczeniu danych w karcie. Wiąże się to z poprawnym projektem aplikacji i wykorzystaniu odpowiednio silnych algorytmów kryptograficznych. Z drugiej strony atakujący może przeprowadzić atak z pominięciem systemu operacyjnego karty (analiza statyczna układu).
— atak właściciela karty przeciw jej wydawcy;
  Ataki te obejmują szereg działań mających na celu kompromitację lub narażenie na straty wydawcę karty. Wydawca może chronić się poprzez aktywny monitoring systemu oraz określenie szczegółowych praw i obowiązków właściciela karty. Przykładowym sposobem ochrony mogą być tzw. czarne listy na których umieszcza się numery zastrzeżonych kart.

— atak właściciela karty przeciw producentowi oprogramowania;
  Celem tego ataku jest modyfikacja oprogramowania w taki sposób aby sprzyjało one określonemu środowisku korzystającemu z kart. Modyfikacja może być wykonana poprzez osobę, która bezpośrednio rozwija dany produkt lub przez hakerów.
— atak właściciela terminala przeciw wydawcy karty;
  Komunikacja pomiędzy posiadaczem karty a jej wydawcą odbywa się poprzez terminal. Możliwość modyfikacji danych protokołu komunikacyjnego jest podstawą tego ataku. Jedną z metod obrony jest stosowanie urządzeń, które swoją budową uniemożliwiają modyfikacje elementów aplikacji. Ponadto komunikacja pomiędzy terminalem a hostem u wydawcy powinna być w odpowiedni sposób zabezpieczona, a jednostki te powinny przed każdą transakcją wzajemnie się uwierzytelnić.
— atak wydawcy przeciw posiadaczom kart;
  Atak ten może polegać na wykorzystaniu kart do np. zbierania informacji o kliencie. Ponadto usługa świadczona przez wydawcę może być sama w sobie oszustwem.
  Posiadacz karty jest praktycznie bezradny. Metodą ochrony może być jedynie rezygnacja z usług wydawcy i ewentualne postępowanie na drodze sądowej.
— atak producenta oprogramowania przeciw posiadaczom kart;
  Jest to celowa modyfikacja oprogramowania na niekorzyść klientów. Obrona przed tym atakiem polega na zewnętrznym audycie oprogramowania.
— inne rodzaje ataków.
  Można do nich zaliczyć np. kradzieże kart, wymuszenia itp.

Różnorodność opisanych powyżej ataków może rodzić pytanie dotyczące sensu inwestowania w karty procesorowe, których celem jest przede wszystkim zwiększenie bezpieczeństwa. Należy jednak pamiętać, że system jest tak bezpieczny jak najsłabsze jego ogniwo, którym w dalszym ciągu, jeśli mówić o systemach informatycznych, jest człowiek. Najwięcej przestępstw i nadużyć z udziałem kart wynika z winy ich właścicieli. Najbardziej wyrazistym tego przykładem jest przechowywanie kodu PIN wraz z kartą. Natomiast ataki polegające na statycznej lub dynamicznej analizie wymagają specjalistycznego sprzętu. Inwestycja taka jest często nieopłacalna w świetle możliwych korzyści z przełamania systemu. Podsumowując - karta potrafi istotnie zwiększyć bezpieczeństwo newralgicznych danych w systemie pod warunkiem, że właściciel odpowiednio o nią dba i w prawidłowy sposób wykorzystuje.

Uwagi bibliograficzne

Dodatkowe informacje związane z atakami na karty procesorowe zawierają publikacje [1, 3].

A. Objaśnienia skrótów

Jedną z trudności jaką napotka czytelnik przy poznawaniu dziedziny związanej z kartami inteligentnymi jest ogromna ilość skrótów. Poniższa lista zawiera wykaz najczęściej spotykanych akronimów.

3DES *triple DES*
ACK *acknowledge*
AES *Advenced Encyryption Standard*
AID *application identifier*
ANSI *American National Standards Institute*
APACS *Association for Payment Clearing Services*
APDU *application protocol data unit*
API *application programming interface*
ASCII *American Standard Code for Information Interchange*
ASN.1 *abstract syntax notation one*
ATR *answer to reset*
BASIC *Beginners All-purpose Symbolic Instruction Code*
BCD *binary coded digit*
BER *basic encoding rules*
BER-TLV *basic encoding rules - tag, length, value*
BTS *base transceiver station*
CA *certification authority*
CAD *chip accepting device*
CAP *card application*
C-APDU *command application protocol data unit*
CBC *cipher block chaining*
CHV *card holder verification*
CLA *class*
CLK *clock*
CMOS *complementary metal oxide semiconductor*
COS *chip operating system*
CPU *central processor unit*
CRC *cyclic redundancy check*
Cryptoki *cryptographic token interface*
CSP *Cryptographic Service Provider*
DEA *data encryption algorithm*
DES *data encryption standard*
DER *distinguished encoding rules*
ECB *electronic code book*
EEPROM *electrically erasable programmable read-only memory*
EF *elementary file*
EFTPOS *electronic fund transfer at point of sale*
EMV *Europay, MasterCard, Visa*
EPROM *erasable programmable read-only memory*
ETSI *European Telecommunications Standards Institute*
etu *elementary time unit*
FAQ *frequently asked questions*
FID *file identifier*
FIFO *first in, first out*
FIPS *Federal Information Processing Standard*
GND *ground*
GNU *General Public License*
GPS *global positioning system*
GPRS *general packet radio service*
GSM *Global System for Mobile Communications*
GUI *graphical user interface*
HSM *hardware security module*
HTML *hypertext markup language*
HTTP *hypertext transfer protocol*
I/O *input/output*
I2C *inter-integrated circuit*
ICC *integrated circuit card*
IDEA *international data encryption algorithm*
IEEE *Institute of Electrical and Electronics Engineers*
IIC *institution identification codes*
IIN *issuer identification number*
IMEI *international mobile equipment identity*
IMSI *international mobile subscriber identity*
IrDA *Infrared Data Association*
ISDN *integrated services digital network*
ISO *International Organization for Standardization*
JVM *Java Virtual Machine*
LAN *local-area network*
Lc *command length*
Le *expected length*
LIFO *last in, first out*

LRC *longitudinal redundancy check*
LSB *least-significant byte*
MAC *message authentication code*
MF *master file*
NPU *numeric processing unit*
NSA *US National Security Agency*
OCF *Open Card Framework*
OS *operating system*
OSI *Open Systems Interconnections*
PC/SC *personal computer/smart card*
PCMCIA *Personal Computer Memory Card International Association*
PDA *personal digital assistant*
PDOL *processing data options list*
PIN *personal identification number*
PIX *proprietary application identifier extension*
PKCS *public key cryptography standards*
PKI *public key infrastructure*
POS *point of sale*
PTS *protocol type selection*
PUK *personal unblocking key*
RAM *random-access memory*
RFC *request for comment*
RFU *reserved for future use*
RID *registered application provider identifier*
RMI *Remote Method Invocation*
ROM *read-only memory*
RSA *Rivest, Shamir and Adleman encryption algorithm*
SAM *secure application module*
SATSA *Security and Trust Services*
SCOS *smart card operating system*
SCQL *structured card query language*
SIM *subscriber identity module*
SMS *short messages service*
SW1, SW2 *status word 1, 2*
TCP *transport control protocol*
TDES *triple DES*
TLV *tag, length, value*
TPDU *transmission protocol data unit*
UART *universal asynchronous receiver transmitter*
USB *Universal Serial Bus*
Vcc *power supply voltage*
VM *virtual machine*
Vpp *programming voltage*
WORM *write once, read multiple*
XML *Extended Markup Language*

B. ATR wybranych kart

Poniżej zamieszczone są ATR (ang. *answer to reset*) wybranych kart.

3B 02 14 50 Schlumberger Multiflex 3k
3B 02 53 01 Gemplus GemClub Memo
3B 17 13 9C 12 02 01 01 07 40 Schlumberger Cyberflex Access Developer 32k
3B 23 00 00 36 41 81 Schlumberger Payflex 4k SAM
3B 23 00 35 11 80 Schlumberger Payflex 1k User
3B 23 00 35 11 81 Schlumberger Payflex 1k SAM
3B 23 00 35 13 FF Schlumberger MicroPayflex
3B 24 00 80 72 94 43 Gemplus MPCOS-3DES 64K (filtr EMV)
3B 27 00 80 65 A2 00 01 01 37 Gemplus GemSAFE Card CSP v1.0
3B 27 00 80 65 A2 02 02 82 37 Gemplus GPK2000s
3B 27 00 80 65 A2 02 03 82 37 Gemplus GPK2000sp
3B 27 00 80 65 A2 04 01 01 37 Gemplus GPK4000s
3B 27 00 80 65 A2 05 01 01 37 Gemplus GPK4000sp
3B 27 00 80 65 A2 0C 01 01 37 Gemplus GPK4000
3B 29 00 80 72 A4 45 64 00 FF 00 10 Gemplus MPCOS-3DES 8K
3B 2A 00 80 65 A2 01 .. .. .. 72 D6 41 Gemplus MPCOS EMV (sektory jednobajtowe)
3B 2A 00 80 65 A2 01 .. .. .. 72 D6 43 Gemplus MPCOS EMV (sektory czterobajtowe)
3B 32 15 00 06 80 Schlumberger Multiflex 8k
3B 32 15 00 06 95 Schlumberger Multiflex 8k DES
3B 63 00 00 36 41 80 Schlumberger Payflex 4k User
3B 65 00 00 9C 11 01 01 03 Schlumberger Cyberflex Palmera
3B 68 00 00 80 66 A2 06 02 01 32 0E Gemplus GemClub 1K
3B 75 94 00 00 62 02 02 0(1,2) 01 Schlumberger Cyberflex 32K e-gate
3B 76 11 00 00 00 9C 11 01 02 02 Schlumberger Cyberflex Access 32K
3B 85 40 20 68 01 01 .. .. Schlumberger Cryptoflex 8k
3B 85 40 20 68 01 01 03 05 Schlumberger Cryptoflex Key Generation
3B 85 40 20 68 01 01 05 01 Schlumberger Cryptoflex 8k
3B 95 18 40 FF 62 01 02 01 04 Schlumberger Cryptoflex 32K e-gate
3B B7 18 00 81 31 FE 65 53 50 4B 32 34 90 00 5A Giesecke & Devrient Starcos 2.4
3B B7 94 00 81 31 FE 65 53 50 4B 32 32 90 00 D0 Giesecke & Devrient Starcos SPK2.2
3B B7 94 00 81 31 FE 65 53 50 4B 32 33 90 00 D1 Giesecke & Devrient Starcos 2.3

Spis tablic

Spis rysunków

Spis wydruków

Książki i artykuły

[1] Adam Shostack Bruce Schneier. *Breaking Up Is Hard To Do: Modeling Security Threads for Smart Cards*, October 1999.
[2] Monika Glinkowska Marian Molski. *Karta elektroniczna – bezpieczny nośnik informacji*. ZNI MIKOM, Warszawa, 1999. Redakcja Joanna Stryjczyk.
[3] W. Effing W. Rankl. *Smart Card Handbook – Second Edition*. WILEY, New York, 1999. Translated by Kenneth Cox.

Standardy i zalecenia

[4] ISO 10202-1:1991. Financial transaction cards – Security architecture of financial transaction systems using integrated circuit cards – Part 1: Card life cycle.
[5] PN-EN ISO/IEC 7813:1999. Karty identyfikacyjne. Karty transakcji finansowych.
[6] PN-ISO/IEC 7810:1997/Ap1:2002. Karty identyfikacyjne. Charakterystyki fizyczne.
[7] ISO/IEC 14443-1:2000. Identification cards – Contactless integrated circuit(s) cards – Proximity cards – Part 1: Physical characteristics.
[8] ISO/IEC 15693-1:2000. Identification cards – Contactless integrated circuit(s) cards – Vicinity cards – Part 1: Physical characteristics.
[9] PN-EN 27816-1:1997. Karty identyfikacyjne. Elektroniczne karty stykowe. Charakterystyki fizyczne.
[10] PN-EN ISO/IEC 10536-1:1998. Karty identyfikacyjne. Elektroniczne karty bezstykowe. Charakterystyki fizyczne.
[11] PN-EN ISO/IEC 7811-1:1997. Karty identyfikacyjne. Technika zapisu. Wytłaczanie.
[12] PN-ISO/IEC 10373-1:2003. Karty identyfikacyjne. Metody badań. Część 1: Badania ogólne.
[13] ISO/IEC 14443-2:2001. Identification cards – Contactless integrated circuit(s) cards – Proximity cards – Part 2: Radio frequency power and signal interface.
[14] ISO/IEC 15693-2:2000/Cor 1:2001. Identification cards – Contactless integrated circuit(s) cards – Vicinity cards – Part 2: Air interface and initialization.
[15] PN-EN 27816-2:1997. Karty identyfikacyjne. Elektroniczne karty stykowe. Wymiary i rozmieszczenie styków.
[16] PN-EN ISO/IEC 7811-2:1997. Karty identyfikacyjne. Technika zapisu. Pasek magnetyczny.
[17] PN-ISO/IEC 10373-2:2003. Karty identyfikacyjne. Metody badań. Część 2: Karty z paskiem magnetycznym.
[18] PN-ISO/IEC 10536-2:1998. Karty identyfikacyjne. Elektroniczne karty bezstykowe. Wymiary i rozmieszczenie obszarów sprzęgających.
[19] ISO/IEC 14443-3:2001. Identification cards – Contactless integrated circuit(s) cards – Proximity cards – Part 3: Initialization and anticollision.
[20] ISO/IEC 15693-3:2001. Identification cards - Contactless integrated circuit(s) cards - Vicinity cards – Part 3: Anticollision and transmission protocol.
[21] PN-EN 27816-3+A:1998. Karty identyfikacyjne. Elektroniczne karty stykowe. Sygnały elektryczne i protokoły transmisji.
[22] PN-EN ISO/IEC 7811-3:1997. Karty identyfikacyjne. Technika zapisu. Rozmieszczenie znaków wytłaczanych na kartach ID-1.
[23] PN-ISO/IEC 10373-3:2004. Karty identyfikacyjne. Metody badań. Część 3: Elektroniczne karty stykowe i powiązane z nimi urządzenia interfejsowe.
[24] PN-ISO/IEC 10536-3:1998. Karty identyfikacyjne. Elektroniczne karty bezstykowe. Sygnały elektroniczne i przełączanie trybów pracy.
[25] ISO/IEC 14443-4:2001. Identification cards – Contactless integrated circuit(s) cards – Proximity cards – Part 4: Transmission protocol.
[26] PN-EN ISO/IEC 7811-4:1997. Karty identyfikacyjne. Technika zapisu. Rozmieszczenie ścieżek magnetycznych tylko do odczytu. Ścieżki 1 i 2.
[27] PN-EN ISO/IEC 7816-4:1998. Technika informatyczna. Karty identyfikacyjne. Elektroniczne karty stykowe. Struktury danych i poleceń komunikacyjnych.

[28] PN-EN ISO/IEC 7811-5:1997. Karty identyfikacyjne. Technika zapisu. Rozmieszczenie ścieżki magnetycznej do zapisu i odczytu. Ścieżka 3.
[29] PN-ISO/IEC 7811-6:1998. Karty identyfikacyjne. Technika zapisu. Pasek magnetyczny wysokiej koercyjności.
[30] ISO/IEC 7816-7:1999. Identification cards – Integrated circuit(s) cards with contacts – Part 7: Interindustry commands for Structured Card Query Language (SCQL).
[31] ISO/IEC 7816-8:2004. Identification cards – Integrated circuit cards – Part 8: Commands for security operations.
[32] ISO/IEC 7816-9:2004. Identification cards – Integrated circuit cards – Part 9: Commands for card management.
[33] EMVCo. *EMV, Integrated Circuit Card Specification for Payment Systems, Book 1, Application Independent ICC to Terminal Interface Requirements, Version 4.1*, 2004.
[34] EMVCo. *EMV, Integrated Circuit Card Specification for Payment Systems, Book 2, Security and Key Management, Version 4.1*, 2004.
[35] EMVCo. *EMV, Integrated Circuit Card Specification for Payment Systems, Book 3, Application Specification, Version 4.1*, 2004.
[36] EMVCo. *EMV, Integrated Circuit Card Specification for Payment Systems, Book 4, Cardholder, Attendant and Acquirer Interface Requirements, Version 4.1*, 2004.
[37] GlobalPlatform. *Card Customization Guide, Version 1.0*, August 2002.
[38] GlobalPlatform. *Guidelines for Developing Java Card Applications on GlobalPlatform Cards, Version 1.0*, December 2002.
[39] GlobalPlatform. *Card Specification, Version 2.1.1*, March 2003.
[40] GlobalPlatform. *Guide to Common Personalization, Version 1.0*, March 2003.
[41] GlobalPlatform. *Load and Personalization Interface Specification, Version 1.0*, March 2003.
[42] GlobalPlatform. *Guidelines for Using GPD 2.0/STIP 2,1 API, Version 1.0*, May 2001.
[43] GlobalPlatform. *Multi Application Smart Card Management System, Functional Requirements, Version 3.4*, May 2001.
[44] GlobalPlatform. *Key Management System, Functional Requirements, Version 1.0*, November 2003.
[45] GlobalPlatform. *A primer to the Implementation of Smart Card Management and Related Systems, Version 1.0*, October 2000.
[46] GlobalPlatform. *Card Specifications 2.1.1 and 2.1 Frequently Asked Questions*, October 2003.
[47] GlobalPlatform. *GlobalPlatform Messaging Specification, Version 1.0*, October 2003.
[48] GlobalPlatform. *System Profiles Specifications, An XML Representation, Version 1.1.0*, September 2003.
[49] GlobalPlatform. *System Scripting Language Specification, An ECMAScript Representation, Version 1.1.0*, September 2003.
[50] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Subscriber Identity Modules (SIM); Functional characteristics (GSM 02.17 version 8.0.0)*, April 2000.
[51] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Test specification for Subscriber Interface Module (SIM) Application Programme Interface (API) for Java card (3GPP TS 11.13 version 8.2.1)*, April 2003.
[52] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Subscriber Identity Module Application Programming Interface (SIM API); Stage 1 (3GPP TS 02.19 version 8.0.0)*, August 2002.
[53] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Security mechanisms for SIM application toolkit; Stage 2 (3GPP TS 03.48 version 8.8.0)*, December 2001.
[54] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface (3GPP TS 11.14 version 8.9.0)*, December 2001.
[55] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); General description of a GSM Public Land Mobile Network (PLMN) (GSM 01.02 version 6.0.1)*, Feburuary 2001.
[56] European Telecommunications Standards Institute. *Digital cellular telecommunications system*

(Phase 2+) (GSM);Specification of the 1.8 Volt Subscriber Identity Module - Mobile Equipment (SIM-ME) interface(GSM 11.18 version 7.0.1), July 1999.
[57] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Security aspects (GSM 02.09 version 8.0.1)*, July 2001.
[58] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface (3GPP TS 11.11 version 8.9.1)*, June 2003.
[59] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2) (GSM); Specification of the 3 Volt Subscriber Identity Module - Mobile Equipment (SIM-ME) interface (GSM 11.12 version 4.3.1)*, March 1998.
[60] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Subscriber Interface Module (SIM) test specification (3GPP TS 11.17 version 8.1.0)*, March 2003.
[61] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2); Abbreviations and acronyms (GSM 01.04 version 8.0.0)*, May 2000.
[62] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Security mechanisms for the SIM application toolkit; Stage 1 (GSM 02.48 version 8.0.0)*, May 2000.
[63] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); GSM API for SIM toolkit stage 2 (3GPP TS 03.19 version 8.5.0)*, September 2002.
[64] RSA Laboratories. *Public Key Cryptography Standard #11 (PKCS #11), Conformance Profile Specification*, 2000.
[65] RSA Laboratories. *Public Key Cryptography Standard #15 (PKCS #15), Conformance Profile Specification*, 2000.
[66] RSA Laboratories. *Public Key Cryptography Standard #15 (PKCS #15), Cryptographic Token Information Syntax Standard, Version 1.1*, 2000.
[67] RSA Laboratories. *Public Key Cryptography Standard #11 (PKCS #11), Cryptographic Token Interface Standard, Version 2.20*, 2004.
[68] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 1, Introduction and Architecture Overview, Revision 1.0*, 1997.
[69] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 2, Interface Requirements for Compatiable IC Cards and Readers, Revision 1.0*, 1997.
[70] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 3, Requirements for PC-Connected Interface Devices, Revision 1.0*, 1997.
[71] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 4, IFD Design Considerations and Reference Design Information, Revision 1.0*, 1997.
[72] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 5, ICC Resource Manager Definition, Revision 1.0*, 1997.
[73] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 6, ICC Service Provider Interface Definition, Revision 1.0*, 1997.
[74] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 7, Application Domain and Developer Design Considerations, Revision 1.0*, 1997.
[75] PC/SC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems, Part 8, Recommendations for ICC Security and Privacy Devices, Revision 1.0*, 1997.

Dokumentacja produktów

[76] Matthias Brüstle. *SCEZ Application Programmer's Manual*, 2000.
[77] CEPSCO. *Common Electronic Purse Specifications, Functional Requirements, Version 6.2*, 1999.
[78] OpenCard Consortium. *OpenCard Framework – General Information Web Document – Second Edition*. IBM Deutschland Entwicklung GmbH, Germany, 1998.
[79] OpenCard Consortium. *OpenCard Framework 1.2 Programmer's Guide*. IBM Deutschland Entwicklung GmbH, Germany, 1999.
[80] David Corcoran. *MUSCLE PC/SC Lite API, Toolkit API Reference Documentation*, 2001.

[81] Microsoft Corporation. *MSDN Documentation*, 2004.
[82] OpenSC Developers. *OpenSC Manual*, 2004.
[83] Andreas Jellinghaus. *OpenCT Manual*, 2004.
[84] Schlumberger. *Cryptoflex Card Reference Manual Version 1.55*, 2000.
[85] Schlumberger. *Cyberflex Access Software Development Kit 4.1, Guide To Schlumberger Smart Card Middleware*, 2001.
[86] Schlumberger. *Cryptoflex Card Programmer's Guide - Cyberflex Access Software Development Kit 4.4*, 2003.
[87] Schlumberger. *Cyberflex Card Programmer's Guide - Cyberflex Access Software Development Kit 4.4*, 2003.
[88] Inc. Sun Microsystems. *Java Card Platform Security, Technical White Paper*. U.S.A., 2001.
[89] Inc. Sun Microsystems. *Security and Trust Services API for Java 2 Platform, Micro Edition, Version 1.0*. U.S.A., 2003.
[90] Inc. Sun Microsystems. *Java Card System Protection Profile Collection, Version 1.0b*. U.S.A., August 2003.
[91] Inc. Sun Microsystems. *Overview of the Java Card System Protection Profile Collection, Version 1.0*. U.S.A., August 2003.
[92] Inc. Sun Microsystems. *Java Card 2.2 Off-Card Verifier, White Paper*. U.S.A., June 2002.
[93] Inc. Sun Microsystems. *RMI Client Application Programming Interface, Java Card 2.2, Java 2 Platform, Micro Edition*. U.S.A., June 2002.
[94] Inc. Sun Microsystems. *Application Programming Interface, Java Card Platform, Version 2.2.1*. U.S.A., October 2003.
[95] Inc. Sun Microsystems. *Application Programming Notes, Java Card Platform, Version 2.2.1*. U.S.A., October 2003.
[96] Inc. Sun Microsystems. *Development Kit User's Guide For the Binary Release with Cryptography Extensions, Java Card Platform, Version 2.2.1*. U.S.A., October 2003.
[97] Inc. Sun Microsystems. *Runtime Environment Specification, Java Card Platform, Version 2.2.1*. U.S.A., October 2003.
[98] Inc. Sun Microsystems. *Virtual Machine Specification, Java Card Platform, Version 2.2.1*. U.S.A., October 2003.

Publikacje i serwisy internetowe

[99] Grupa dyskusyjna poświęcona kartom inteligentnym. `alt.technology.smartcards`.
[100] Karty bankowe i bankomaty. `http://www.karty.pl/`.
[101] MULTOS. `http://www.multos.com/`.
[102] M.U.S.C.L.E. `http://www.linuxnet.com/`.
[103] Najczęściej zadawane pytania na forum dyskusyjnym `alt.technology.smartcards`. `http://www.mobile-mind.com/htm/scfaq.htm`.
[104] Phoenix. `http://joshyfun.webzdarma.cz/phoenix/`.
[105] Świat Kart Płatniczych. `http://www.kartyonline.net/`.
[106] Alexandre Becoulet. Smart Card ToolKit. `http://diaxen.free.fr/sctk/`.
[107] Matthias Bruestle. SOSSE Homepage – Simple Operating System for Smartcard Education. `http://www.mbsks.franken.de/sosse/`.
[108] Wolfgang Rankl. Homepage. `http://www.wrankl.de`.