

Stefan A. Nawrocki

**System
komputerowego
składu
dokumentów**

Kombi v. 8

1. Jak zacząć

2. Kombi

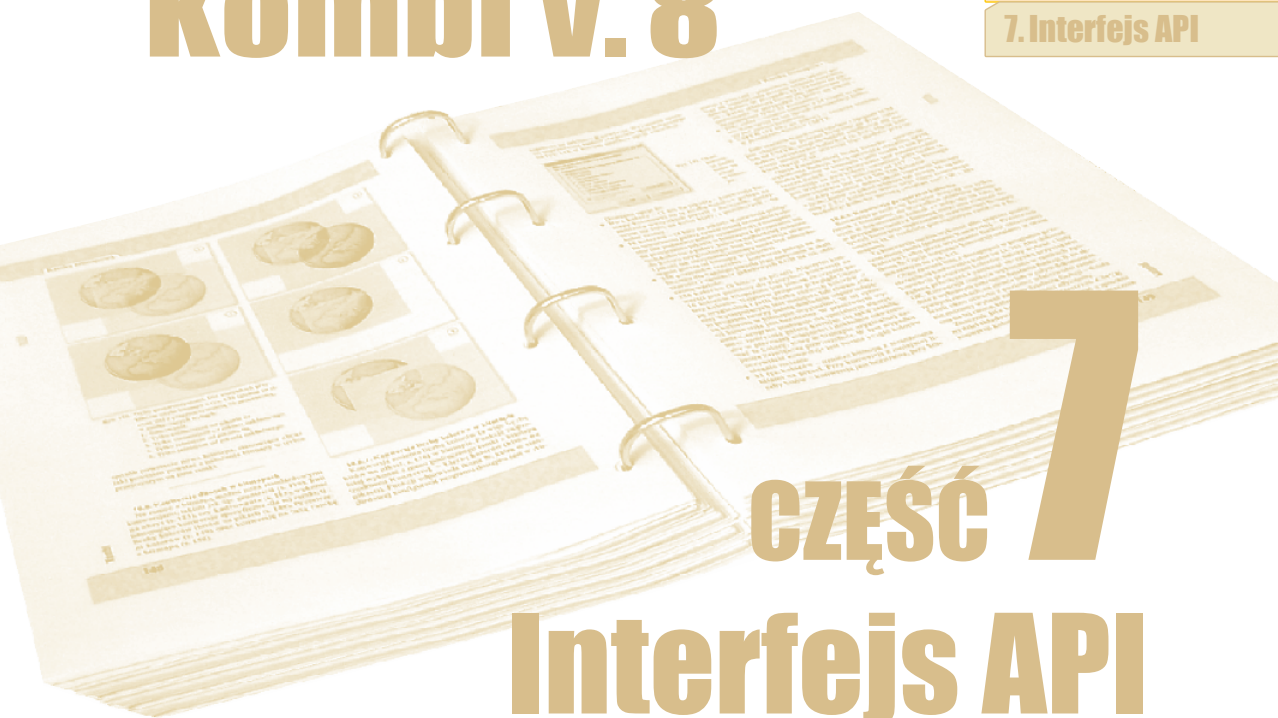
3. KombiKor

4. Filtry rastrowe

5. Przykłady

6. Katalog grafik

7. Interfejs API



CZĘŚĆ 7 Interfejs API



Stefan A. Nawrocki

KOMBI v. 8

Interfejs API

Opracowanie graficzne:

3N, 1996-2007 za pomocą programu KOMBI

Produkcja, dystrybucja i serwis:

3N, Usługi Komputerowe

75-814 Koszalin

ul. Zakole 20/7

www.3n.com.pl

e-mail: office@3n.com.pl

tel. 502 362 175

Copyright by 3N Usługi Komputerowe 1996-2007.

Żadna część niniejszej publikacji nie może być w jakiegokolwiek formie kopiowana bez pisemnej zgody właściciela praw autorskich tj. firmy **3N Usługi Komputerowe w Koszalinie**.

Wszelkie nazwy handlowe i towarów występujące w tym opracowaniu są znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odnośnych właścicieli.

Portions of this software are based in part on the work of the Independent JPEG Group.

The Graphics Interchange Format (c) is the Copyright property of CompuServe Incorporated. GIF (sm) is a Service Mark property of CompuServe Incorporated.

Producent dokłada wszelkich starań, aby informacje zawarte w dokumentacji były aktualne. Jednak, ze względu na ciągły rozwój oprogramowania, zastrzega się możliwość zmiany działania niektórych funkcji, co w konsekwencji spowoduje nieaktualność niektórych informacji zawartych w dokumentacji. Zaleca się w szczególności zapoznanie z treścią plików **Co nowego?** oraz **Notami technicznymi**. Pliki te dostępne są z poziomu programów: Kombi i KombiKor (menu **Pomoc**).

Spis treści

1. Wstęp	7
2. Interfejs API Kombi	9
2.1. Instalowanie rozszerzeń.....	9
2.2. Licencjonowanie rozszerzeń	10
2.3. Rozszerzenia typu kxm	10
2.4. Rozszerzenia typu exe	11
2.5. Obsługa przycisku „Informacje do- datkowe	12
2.6. Filtry rastrowe	12
2.7. Filtry wejścia/wyjścia	17
2.8. Komunikacja z programem głównym rozszerzeń typu exe	18
3. Indeksy	21
3.1. Indeks rzeczowy	21

1. Wstęp

Ten dokument jest częścią siódmą („brązową”) dokumentacji pakietu Kombi i poświęcony jest interfejsowi API (application programming interface) tego pakietu. Pod tym pojęciem rozumiem zestaw struktur, funkcji i stałych pozwalających tworzyć zewnętrzne rozszerzenia wzbogacające możliwości programu.

Wraz z tą dokumentacją dostarczany jest plik **kombi_8_api_exe**, który zawiera przykładowe pliki oraz plik **x_def.h** z definicjami struktur, funkcji i stałych potrzebnych do tworzenia rozszerzeń pakietu.

Zakłada się, że użytkownik chcący tworzyć własne rozszerzenia ma opanowane programowanie w języku C i C++ w stopniu przynajmniej dobrym. Przykłady zostały przygotowane w kompilatorze firmy Borland (v. 4.52). Wersja ta jest dostępna nieodpłatnie do testów na CD-ROM-ie dołączonym do czasopisma PC World Komputer (maj/98). Oczywiście jest możliwe tworzenie rozszerzeń w dowolnym innym kompilatorze, ale w takim przypadku należy stworzyć samodzielnie projekt i dowiązać do niego pliki znajdujące się w katalogach z przykładami.

Katalog z przykładami zostanie automatycznie utworzony po zainstalowaniu wspomnianego wcześniej pliku i będzie nazywał się **api**. Wewnątrz niego znajdziemy następujące podkatalogi:

- **api_kxm** – przykład z podstawowymi funkcjami obsługującymi instalowanie i licencjonowanie rozszerzeń typu **kxm**,
- **api_exe** – jak wyżej, ale dla rozszerzeń typu **exe**,
- **filtr** – przykładowy (w pełni działający) filtr rastrowy,

- **io** – przykładowy filtr wejścia/wyjścia. Filtr nie importuje rzeczywistych bitmap, ale „udaje” import przez losowe wypełnianie pikseli,
- **cmd_list** – przykładowe w pełni działające rozszerzenie typu **exe**. Rozszerzenie tworzy drzewo komend dostępnych w Kombi i umożliwia wykonywanie tych komend poprzez dwukrotne kliknięcie w wybraną komendę.

We wszystkich omówionych wyżej katalogach znajdują się kody źródłowe oraz skompilowane pliki gotowe do zainstalowania jako rozszerzenia.

Niniejsza publikacja jest fragmentem pełnej dokumentacji pakietu i w wielu miejscach będę się odwoływał do innych jej części. W takim przypadku, numer strony do której się odwołuję będzie poprzedzony odpowiednią ikoną, np. odwołanie do pliku głównego dokumentacji (części „niebieskiej” poświęconej programowi Kombi) – ikoną 📄. Opis wszystkich oznaczeń stosowanych w dokumentacji znajduje się w części pierwszej (**Jak zacząć**, s. 7).

Pełną wersję dokumentacji w postaci plików *.pdf, samorozpakowujących się archiwów aktualizujących pakiet, a także – dokumentów źródłowych złożonych w Kombi (*.kme) można pobrać z naszego serwera internetowego – www.3n.com.pl.

Opis procedury instalowania Kombi w wersji pełnej z dysku instalacyjnego, a także – wersji demo oraz aktualizacji pobieranych przez Internet znajduje się w **Podręczniku użytkownika** (📄 s. 15).

Wszelkie uwagi na temat niniejszej dokumentacji (jak i całego programu) proszę kierować na adres:

3N Usługi Komputerowe
office@3n.com.pl

Zapraszam również do korzystania z naszego forum dyskusyjnego – www.3n.com.pl/forum.php.



Zapraszam wszystkich chętnych do wspólnego rozwijania pakietu. Mam nadzieję, że w ten sposób będzie on mógł rozwijać się jeszcze szybciej, zapewniając i użytkownikom i twórcom rozszerzeń wiele satysfakcji.

2. Interfejs API Kombi

Rozszerzenia zewnętrzne są bibliotekami dynamicznymi (plikami *.dll, którym zmieniono rozszerzenie na *.kxm) lub plikami wykonywalnymi (*.exe). Typ pierwszy w związku z tym będę nazywał typem kxm, zaś drugi – typem exe. Z programowego punktu widzenia różnica między nimi polega na tym, że biblioteka dynamiczna po załadowaniu staje się częścią programu. Z jednej strony umożliwia to swobodniejszy dostęp do danych w programie, ale z drugiej – krytyczny błąd w bibliotece spowoduje załamanie całego programu. W przypadku plików exe – usługa dostarczana przez ten plik jest wykonywana w osobnym procesie. Dane między programem głównym, a programem usługowym są przesyłane w sposób dużo bardziej złożony niż w przypadku biblioteki, ale – załamanie się programu usługowego nie ma wpływu na stabilność pracy programu głównego.

W rozdziale omawiam tworzenie, instalowanie i licencjonowanie zewnętrznych rozszerzeń programu Kombi.

2.1. Instalowanie rozszerzeń

Bez względu na typ rozszerzenia, każde z nich musi być przed uruchomieniem zainstalowane w programie głównym. Z punktu widzenia użytkownika, ta instalacja polega na „wczytaniu” rozszerzenia. Możemy to zrobić z poziomu skrzynki **Rozszerzenia**, którą utworzymy z menu **Rozszerzenia** → **Instaluj**. Po otwarciu okna **Rozszerzenia**, znajdziemy w nim odnośnik **doinstalować**. Kliknięcie odnośnika otwiera okno wyboru pliku, poprzez które „wczytujemy” wybrane rozszerzenie.

Oczywiście jest to metoda na ręczne doinstalowanie np. nowego (testowanego) rozszerzenia. Rozszerzenia „firmowe” są instalowane automatycznie po kliknięciu odnośnika **wyszukać i zainstalować automatycznie**. Ta sama procedura automatycznego instalowania jest również wykonywana po każdej aktualizacji i polega ona na przejrzaniu pliku **kombi_xtension.ini**, który znajduje się w tym samym katalogu, co plik Kombi.exe.

Plik **kombi_xtension.ini** jest plikiem tekstowym, a jego strukturę omówię na przykładzie kilku wpisów:

```

————— (fragment pliku kombi_xtension.ini) —————
1. KOMBI_XTENSION_LIST
2. Compress.kxm=Kompresor/Dekompresor danych (wymaga załadowania); 1; COMPRESSOR
3. Dwuton.kxm=Filtr rastrowy (wymaga załadowania); 1
4. Kreator_palet.exe=Kreator palet narzędziowych (wymaga załadowania); 1; Kreator_palet
————— (koniec fragmentu pliku kombi_xtension.ini) —————

```

- **KOMBI_XTENSION_LIST** – nagłówek pliku;
- **Compress.kxm** – nazwa pliku stanowiącego rozszerzenie;
- **Kompresor/Dekompresor danych** (wymaga załadowania) – komentarz, który jest wyświetlany w oknie wyboru pliku, kiedy użytkownik wybierze w tym oknie dane rozszerzenie.
- **1** – wartość wskazująca, czy podczas automatycznego instalowania rozszerzeń dane rozszerzenie zostanie załadowane, czy nie. Jeśli wartość ta wynosi 1, rozszerzenie zostanie załadowane, jeśli 0 – nie.
- **COMPRESSOR** – nazwa wewnętrzna rozszerzenia. Nazwa ta wykorzystywana jest w kilku sytuacjach. Po pierwsze, jeśli rozszerzenie jest samodzielnym programem, to jest to jednocześnie nazwa klasy głównego okna rozszerzenia. Program główny podczas zamykania wykonuje dla każdego rozszerzenia funkcję **FindWindow** z nazwą wewnętrzną tego rozszerzenia. Jeśli funkcja **FindWindow** zwróci uchwyt, tzn., że rozszerzenie jest aktywne i wtedy program główny wysyła do tego okna komunikat **WM_CLOSE**, powodujący jego zamknięcie.

Po drugie – nazwa klasy jest wykorzystywana do związania danej usługi z funkcją, a także wyświetlania komunikatów informujących o braku jakiegoś rozszerzenia. Np. jeśli dokument jest skompresowany, a podczas próby otwarcia tego dokumentu kompresor/dekompresor nie będzie załadowany, program przejrzy zawartość omawianego pliku i znajdzie, że rozszerzenie klasy **COMPRESS** jest obsługiwane przez plik **Compress.kxm**, po czym wyświetli komunikat informujący użytkownika, że brakuje rozszerzenia **Compress.kxm**.

W przypadku filtrów rastrowych (przykładowy wpis **Dwuton.kxm**) nazwa klasy nie jest wymagana.

Nazwa klasy jest ponadto zdublowana wewnątrz (w kodzie) danego rozszerzenia, tak więc wpis w pliku **kombi_xtension.ini** nie jest konieczny do pracy rozszerzenia, a umożliwia jedynie jego automatyczne załadowanie oraz ewentualne powiadomienie użytkownika, którego rozszerzenia brakuje (choć nazwa klasy jest zdublowana w rozszerzeniu, to w przypadku

jego braku – oczywistym jest, że bez wpisu w pliku **kombi_xtension.ini** program nie będzie mógł podać nazwy dyskowej brakującego pliku. W tej sytuacji program w komunikacie informującym o braku rozszerzenia podaje nazwę klasy). W przypadku braku filtra rastrowego potrzebnego do otwarcia dokumentu, program poinformuje użytkownika komunikatem, w którym podaje nazwę brakującego filtra (nazwa ta jest pamiętana w dokumencie).

Tak więc, zaleca się, aby po stworzeniu rozszerzenia, które miałyby funkcjonować na rynku, zgłosić to do producenta, który dokona odpowiedniego wpisu w dystrybuowanym pliku **kombi_xtension.ini**. Trzeba tu też dodać, że obecność wpisu w tym pliku nie skutkuje żadnymi konsekwencjami w przypadku nie znalezienia danego rozszerzenia na dysku. Tzn. w pliku są wyszczególnione wszystkie możliwe rozszerzenia, ale jeśli użytkownik nie nabeździe któregoś z nich, to dany wpis będzie po prostu ignorowany.

„Wczytanie” rozszerzenia powoduje dopisanie go do listy rozszerzeń aktywnych i zapisanie tego faktu w pliku konfiguracyjnym, co spowoduje, że po ponownym uruchomieniu programu będzie ono instalowane (wczytywane) automatycznie.

Nie jest wymagane ponowne uruchomienie programu po ręcznym dodaniu lub usunięciu rozszerzenia.

2.2. Licencjonowanie rozszerzeń

Na czym polega licencjonowanie rozszerzenia? Interfejs API Kombi przewiduje możliwość związania numeru licencji rozszerzenia z numerem licencji programu. Użytkownik nabywając pakiet otrzymuje numer licencyjny (tzw. „długi”), który wpisuje na etapie instalowania programu (w oknie instalatora). Jeśli niezależna firma stworzy własne rozszerzenie programu, to aby je zainstalować, użytkownik podaje producentowi pięć pierwszych znaków tego numeru. Załóżmy, że numer licencji jest 141CCA594D. Użytkownik nabywając rozszerzenie u niezależnego producenta podaje numer 141CC. Producent rozszerzenia przetwarza ten numer w wiadomy tylko sobie sposób (wylicza znanym sobie algorytmem tzw. sumę kontrolną) i podaje ją użytkownikowi. W przypadku algorytmu zawartego w przykładowych plikach byłby to numer 1230. Użytkownik wprowadza ten numer w okienku instalowania danego rozszerzenia. Rozszerzenie podczas pierwszego uruchomienia otrzymuje z programu numer „długi” licencji. Algorytm wyliczający sumę kontrolną musi być zaszyty w kodzie rozszerzenia i po otrzymaniu numeru licencji algorytm ten wylicza sumę kontrolną i sprawdza ją z numerem podanym przez użytkownika. Jeśli jest zgodność obu numerów, rozszerzenie zapisuje wprowadzony numer licencji na dysku (w pliku *.lic) co spowoduje, że

podczas kolejnego uruchomienia użytkownik nie będzie musiał tego numeru podawać.

Wykorzystanie do generowania sumy kontrolnej pierwszych pięciu cyfr numeru licencji zapewnia z jednej strony wystarczająco dużą liczbę kombinacji, aby uniemożliwić przypadkowe „odkrycie” właściwego numeru, z drugiej zaś – chroni użytkownika i producenta pakietu przed ujawnieniem pełnego numeru licencji, który to numer wykorzystywany jest do pobierania bezpłatnej aktualizacji.

2.3. Rozszerzenia typu kxm

Każde rozszerzenie komunikuje się z programem głównym poprzez nagłówek typu KombiXtensionHeader. Jest on zdefiniowany następująco:

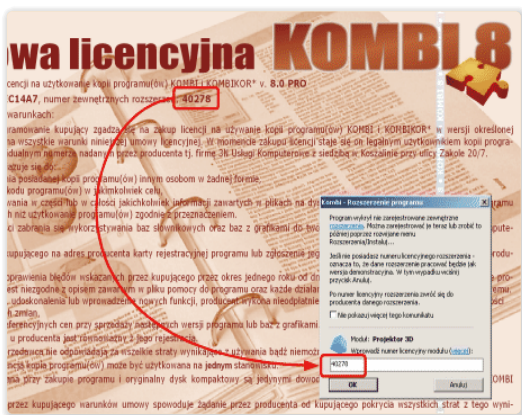
```
typedef struct{
    ATOM name;
    ATOM firm;
    ATOM version;
    ATOM className;
    int type;
    int status;
    XT_GetInfo* getInfo;
    XT_GetLicence* getLicence;
    XT_SetLicence* setLicence;
} KombiXtensionHeader;
```

- name – nazwa rozszerzenia;
- firm – nazwa firmy produkującej rozszerzenie,
- version – numer wersji rozszerzenia;
- className – nazwa klasy (o której pisałem wyżej);
- type – typ rozszerzenia. Dostępne są następujące typy:

#define KOMBI_XT_TYPE_RASTER_FILTER	0
//filtr rastrowy,	
#define KOMBI_XT_TYPE_MODULE	1
//moduł (aktualnie nie wykorzystane),	
#define KOMBI_XT_TYPE_SERVICE	2
//usługa,	
#define KOMBI_XT_TYPE_SERVICE_NOOPT	3
//usługa, która nie wymaga okna do ustalania	
//dodatkowych opcji (czyli usługa bez opcji),	
#define KOMBI_XT_TYPE_IO_FILTER	4
//filtr wejścia/wyjścia;	
- status – w tym polu program przechowuje bitowo stan, w którym znajduje się rozszerzenie. Możliwe są następujące stany:

#define KOMBI_XT_STATUS_LOAD	1
//oznacza, że rozszerzenie jest załadowane,	
#define KOMBI_XT_STATUS_LICENCE	2
//oznacza, że program sprawdził już stan licencji	
//dla tego rozszerzenia i numer licencyjny tego	
//rozszerzenia jest prawidłowy,	

- `getInfo` – wskaźnik do funkcji typu `XT_GetInfo`, którą zadeklarowano następująco: `typedef void (XT_GetInfo)(HWND)`, co oznacza, że funkcja przyjmuje jeden parametr typu `HWND` i nie zwraca wartości. Funkcją jest wykorzystywana do wyświetlania informacji o rozszerzeniu. Jeśli użytkownik otworzy okno właściwości danego rozszerzenia i wciśnie przycisk **Informacje dodatkowe**, to program wywoła tę funkcję, która może np. otworzyć okno dialogowe z opisem rozszerzenia lub też – odpowiedni plik pomocy.
- `getLicence` – wskaźnik do funkcji typu `XT_GetLicence`, którą zadeklarowano jako: `typedef char* (XT_GetLicence)()`, co oznacza, że funkcja nie pobiera żadnych parametrów, natomiast zwraca napis. Zwracany napisem jest string zawierający numer licencyjny rozszerzenia. Jeśli użytkownik nie wprowadził numeru licencyjnego rozszerzenia, funkcja zwraca łańcuch pusty.
- `setLicence` – wskaźnik do funkcji typu `XT_SetLicence`, którą zadeklarowano jako: `typedef void (XT_SetLicence)(HWND hWnd, char* XtensionCopyNum)`, co oznacza, że funkcja przyjmuje dwa parametry i nie zwraca żadnego. Funkcja służy do wprowadzenia numeru licencji i jest wykonywana po wciśnięciu przycisku **OK** w oknie pokazanym na **rys. 1**. Pierwszy przyjmowany parametr, to uchwyt do okna pokazanego na tym rysunku, natomiast drugi – to wskaźnik do napisu edytowanego w tym samym oknie.



Rys. 1. Okno wprowadzania numeru licencyjnego rozszerzeń.

Dla rozszerzeń typu `kxm`, w celu utworzenia nagłówka wykonywana jest funkcja `extern "C" KombiXtensionHeader* CALLBACK InitXtension(char* path, char* KombiCopyNum)`, do której Kombi przekazuje ścieżkę do danego rozszerzenia oraz numer licencyjny programu.

W katalogu `katalog_programu\wtyczki\api\api_kxm` znajdziemy przykładowy kod źródłowy rozszerzenia, które nie wykonuje żadnej usługi, ale obra-

zuje mechanizm rejestrowania i licencjonowania rozszerzeń typu `kxm`.

Na ten kod składa się siedem plików:

- `api_kxm.ide` – jest to projekt programu przygotowany w Borlandzie v. 4.52,
- `api_kxm.cpp` – kod źródłowy rozszerzenia,
- `api_kxm.h` – plik nagłówkowy do pliku `api_kxm.cpp`,
- `api_kxm.rc` – plik z zasobami,
- `api_kxm.rh` – plik nagłówkowy do pliku z zasobami,
- `api_kxm.def` – plik opisujący parametry tworzonej biblioteki,
- `x_def.h` – plik definiujący struktury i stałe.

Aby skompilować projekt, należy uruchomić kompilator i otworzyć projekt. Jeśli nie dysponujemy wymienioną wyżej wersją kompilatora, to można utworzyć w dowolnym kompilatorze nowy projekt i w opcjach projektu ustalić, że kompilowana będzie biblioteka dynamiczna, a następnie „podłączyć” do projektu wymienione wyżej pliki (poza `api_kxm.ide`).

Następnie wykonujemy polecenie **Zbuduj** (Build node). Po skompilowaniu powinien powstać plik `api_kxm.dll`, który przemianowujemy na `api_kxm.kxm`. Teraz możemy otworzyć okno **Rozszerzenia** (w programie Kombi menu **Rozszerzenia** → **Instaluj**) i kliknąć odnośnik **doinstalować**. Spowoduje to otwarcie okna wyboru pliku, w którym odnajdujemy utworzoną przed chwilą bibliotekę. Po jej dwukrotnym kliknięciu skompilowane przed chwilą rozszerzenie pojawi się na liście rozszerzeń w oknie **Rozszerzenia**, a także – w menu **Kombi** → **Programy**.

Kliknięcie w nazwę rozszerzenia spowoduje otwarcie jego okna właściwości. Zobaczmy jak działa przycisk **Dodatkowe informacje i Licencja**. Stworzmy numer licencyjny naszego rozszerzenia wg algorytmu opisanego w pliku `pusty.cpp` i zarejestrujemy nim rozszerzenie. Jeśli wszystko się powiedzie znak **!** powinien zniknąć, co oznacza, że rozszerzenie jest prawidłowo zainstalowane i zarejestrowane.

2.4. Rozszerzenia typu exe

W katalogu `katalog_programu\wtyczki\api\api_exe` znajdziemy kod podobny do omówionego wyżej, ale dla rozszerzenia będącego plikiem wykonywalnym.

Różnica między tym kodem, a poprzednim polega na innej obsłudze nagłówka `KombiXtensionHeader`. W rozszerzeniu typu `kxm` pamięć dla nagłówka jest przydzielana przez rozszerzenie. W rozszerzeniu typu `exe` – pamięć dla nagłówka jest przydzielana w programie głównym, który „wystawia” tę pamięć w pliku wymiany. Podczas pierwszego uruchomienia rozszerzenia w linii komend przesyłane jest polecenie „InitXtension”, a za nim (po spacji) numer licencji („długi”) programu. Rozszerzenie sprawdza,

czy w linii komend jest polecenie „InitXtension” i jeśli tak – to tworzy nagłówek lokalny, który po wypełnieniu jest przesyłany pod „wystawiony” przez program główny adres.

Funkcje GetLicence() i SetLicence() działają podobnie jak dla rozszerzeń **kxm**, ale ich adresy nie są pamiętane w nagłówku (ponieważ są one wykonywane w innym procesie). Zamiast tego – rozszerzenie musi obsługiwać dwa komunikaty (IDM_KOMBI_SET_LICENCE i IDM_KOMBI_GET_LICENCE), poprzez które program główny ustawia i odczytuje licencję rozszerzenia.

Proponuję przeanalizować przykładowy kod znajdujący się we wspomnianym wyżej katalogu.

2.5. Obsługa przycisku „Informacje dodatkowe”

W przypadku rozszerzeń **kxm**, twórca rozszerzenia może napisać funkcję typu void About(HWND), wewnątrz której zawarty powinien być kod (np. MessageBox) wyświetlający informacje o rozszerzeniu. Nazwę funkcji należy wstawić do pola getInfo w nagłówku rozszerzenia.

W przypadku rozszerzeń typu **exe** takie rozwiązanie nie jest dostępne.

W obu jednak sytuacjach dostępne jest inne (zalecane) rozwiązanie polegające na wykorzystaniu plików typu *.khp. Pliki tego typu sterują wyświetlaniem plików *.rtf w Eksploratorze zasobów. Każdy plik *.khp tworzy w Eksploratorze zasobów odrębną gałąź. Wewnątrz pliku zawarta jest nazwa gałęzi oraz informacje dodatkowe, np. czy gałąź zawiera podgałęzie, skąd pobrać ikonę gałęzi, itp. Po wybraniu danej gałęzi, program odczytuje te informacje i w prawym panelu okna Eksploratora wyświetla plik *.rtf o nazwie ustalonej na podstawie danych zawartych w pliku *.khp.

Pliki *.khp są plikami tekstowymi. Przykład pliku *.khp znajduje się w katalogu **katalog_programu\api\api_exe** i niżej omówię strukturę plików tego typu na tym przykładzie.

(plik api_exe.khp)

1. ;Przykładowe rozszerzenie
2. ;api_exe_
3. ;api_exe.rtf
4. ;wtyczki\api_exe\api_exe.exe
5. ;%xtension
6. Podstrona_1
7. Podstrona_2

(koniec pliku api_exe.khp)

- ;Przykładowe rozszerzenie – nazwa, która będzie nazwą gałęzi w Eksploratorze zasobów;

- ;api_exe_ – prefiks, do którego będą dodawane kolejne cyfry w celu utworzenia nazw plików *.rtf stanowiących treść opisów podstron; np. jeśli gałąź zawiera dwie podgałęzie, to będą to pliki: api_exe_0.rtf i api_exe_1.rtf; jeśli gałąź nie zawiera podgałęzi, to pole jest ignorowane;
- ;api_exe.rtf – nazwa pliku rtf, w którym jest opis strony głównej rozszerzenia;
- ;wtyczki\api_exe\api_exe.exe – nazwa pliku wykonywalnego, z którego pierwsza ikona będzie pobrana do oznaczenia gałęzi w Eksploratorze zasobów;
- ;%xtension – nazwa gałęzi, w której pojawi się gałąź opisywana danym plikiem *.khp;
- Podstrona_1 – nazwa pierwszej podstrony;
- Podstrona_2 – nazwa drugiej podstrony.

Jeśli gałąź opisywana plikiem *.khp ma więcej podstron, to w kolejnych wierszach należy podać ich nazwy. Jednocześnie należy w tym samym katalogu co dany plik *.khp utworzyć odpowiednią liczbę plików *.rtf o nazwach składających się z prefiksu (w naszym wypadku **api_exe_** i kolejnych cyfr, np. **api_exe_0.rtf**, **api_exe_1.rtf**, itd.

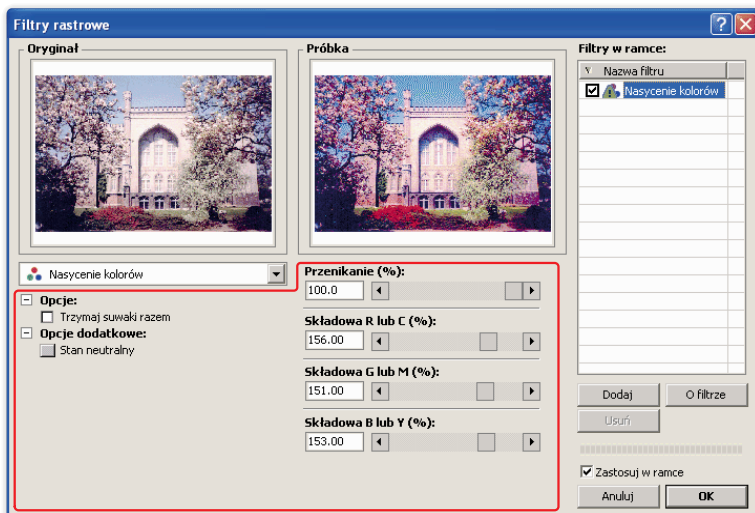
Jeśli gałąź nie zawiera podstron, to ostatnim wierszem w pliku *.khp jest wiersz %xtension.

W każdym wypadku, w tym samym katalogu musi się znaleźć plik *.rtf wyszczególniony w wierszu 3 pliku *.khp. Plik *.rtf zawierający opis rozszerzenia nie musi spełniać jakichś specjalnych wymagań. Może to być dowolny plik *.rtf utworzony w dowolnym programie. Jednak może on zawierać dodatkowe „tagi”, które będą przez Kombi rozumiane i obsługiwane w specyficzny sposób. Myślę, że najprościej będzie utworzyć własny plik *.rtf przez edycje przykładowych plików zawartych w katalogu **katalog_programu\api\api_exe**.

Przycisk **Informacje dodatkowe** na karcie właściwości rozszerzenia utworzy przygotowany przez nas plik *.khp, jeśli nazwa tego pliku jest taka sama jak nazwa dyskowa danego rozszerzenia (oczywiście z rozszerzeniem khp) i pole getInfo w nagłówku jest równe zero. W przypadku rozszerzeń typu **kxm** i wypełnienia pola getInfo adresem funkcji About – program wykona tę funkcję. W przypadku rozszerzeń typu **exe**, pole getInfo nie ma znaczenia. Jeśli pole getInfo jest równe zero i w katalogu z rozszerzeniem nie ma odpowiedniego pliku *.khp, przycisk **Dodatkowe informacje** będzie nieaktywny.

2.6. Filtry rastrowe

Z punktu widzenia użytkownika filtry rastrowe obsługiwane są poprzez okno filtrów rastrowych pokazane na **rys. 2**. W oknie wyróżnić można elementy stałe (niezależne od rodzaju filtru) i zmienne, których obecność zależy od rodzaju filtru. Elementy stałe, to: w górnej części dwa okna z podglądem (lewe pokazuje



Rys. 2. Okno filtrów rastrowych. Czerwoną linią otoczono elementy zależne od wybranego aktualnie filtru.

bitmapę przed zastosowaniem filtru, zaś prawe – po zastosowaniu), pod podglądami – rozwijana lista wyboru typu filtru, w prawej części – lista filtrów przypisanych ramce oraz pod nią – kilka przycisków wspomagających zarządzanie filtrami.

Elementy zmienne – otoczono na **rys. 2** czerwoną linią. Obecność tych elementów zależy od rodzaju filtru i jest (poza dwoma wyjątkami) definiowana w samym filtrze. Wspomniane wyjątki, to:

- Pole wyboru nazwane **Ścieżka odcinania**. Obecność tego elementu zależy od tego, czy w ramce jest zdefiniowana ścieżka odcinania, czy nie. To pole wyboru jest dodawane (i obsługiwane) automatycznie przez program do każdego filtru.
- Suwak **Przenikanie**. Ten suwak jest zawsze dodawany automatycznie do każdego filtru i również jest automatycznie obsługiwany.

Dodatkowo – wewnątrz obszaru zaznaczonego na **rys. 2** czerwoną linią – mamy opcjonalnie następujące elementy interfejsu:

- Grupę opcji nazwaną **Opcje dodatkowe**. Zarówno sama nazwa grupy opcji, jak i nazwy poszczególnych opcji są definiowane w filtrze. Filtr może zawierać do 32 opcji tego typu. Są to opcje typu włącz-wyłącz i w dalszym opisie będą nazywał je opcjami typu **checkbox**.
- Pole wyboru na rysunku nazwane **Podtyp**. Jak wyżej – nazwa ta, jak i nazwy poszczególnych pól wyboru – są definiowane w filtrze. Liczba pól wyboru jest dowolna. W dalszym opisie te grupy opcji będą nazywał grupą **radiobutton**.
- Grupa przycisków nazwana w przykładzie **Funkcje dodatkowe**. Nazwa grupy, jak i nazwy poszczególnych funkcji są definiowane w filtrze. Liczba przycisków (funkcji) jest dowolna. Wykonanie

funkcji nastąpi po wciśnięciu myszką prostokątnego przycisku przed nazwą funkcji. Ten element interfejsu będzie dalej nazywał grupą **button**.

Filtrы rastrowe zbudowane są na bazie rozszerzeń typu **kxm**. W katalogu `katalog_programu\api\filtr` znajdziemy przykładowy filtr rastrowy. Poza opisanym wyżej nagłówkiem `KombiXtensionHeader` w przypadku filtrów rastrowych zastosowanie mają opisane niżej struktury.

```
struct rasterFilterData{
    XT_FilterFunction* filterFunction;
    XT_AddOption* addOption;
    XT_SetData* setData;
    XT_GetData* getData;
    XT_SetPar* setPar;
    XT_GetPar* getPar;
    rasterFilterPar par[8];
    int indexName;
    int nOptCheck;
    int indexOptCheck;
    int nOptRadio;
    int indexOptRadio;
    int nPar;
    int indexPar;
    int nAddOption;
    int indexAddOption;
};
```

- `filterFunction` – jest adresem do funkcji typu `XT_FilterFunction`. Funkcję tę zdefiniowano jako: `typedef int (XT_FilterFunction)(HWND hWnd, long w, long h, void* bits, int check, int radio, float* par, int bps)`, gdzie:
 - `hWnd` jest uchwytym do paska postępu w oknie filtrów rastrowych,
 - `w` jest szerokością bitmapy w pikselach,
 - `h` jest wysokością bitmapy w pikselach,

- bits jest adresem danych w bitmapie,
- check jest zmienną, w której ustawienie kolejnych bitów odpowiada włączeniu opcji typu checkbox w oknie filtrów rastrowych,
- radio jest zmienną, która odpowiada wybranej opcji typu radiobutton w oknie filtrów rastrowych,
- par jest wskaźnikiem do tablicy parametrów (maksymalnie ośmiu) odpowiadających parametrom ustawionym na suwakach w oknie filtrów rastrowych,
- bps (bits per sample) liczba bitów na próbkę, która może wynosić 24 (bitmapa RGB) lub 32 (bitmapa CMYK).
Funkcja zwraca wartość 1, jeśli filtr został zastosowany lub 0, jeśli nie mógł być zastosowany (np. z powodu niezarejestrowania rozszerzenia);
- addOption jest wskaźnikiem do funkcji odpowiadającej na zdarzenia pochodzące od opcji typu **button** w oknie filtrów rastrowych. Zdefiniowano ją jak następuje: typedef int (XT_AddOption)(RasterFilter* rf, HWnd hWnd, int function), gdzie:
 - rf jest wskaźnikiem do obiektu typu RasterFilter (omówionego dalej),
 - hWnd jest uchwytem do okna dialogowego filtrów rastrowych,
 - function jest liczbą odpowiadającą użytemu przyciskowi. Wartość 1 oznacza, że użyto pierwszego przycisku, 2 – że drugiego, itd.;
- getData jest wskaźnikiem do funkcji zapisującej dane (parametry) filtru w pliku. Jest ona zdefiniowana jako typedef int (XT_GetData)(RasterFilter* rf, void*), gdzie:
 - rf jest wskaźnikiem do obiektu typu RasterFilter,
 - p jest wskaźnikiem do bloku pamięci.
Działanie tej funkcji polega na tym, że kiedy program Kombi zapisuje dokument na dysku i w dokumencie jest użyty filtr, to program przekazuje do filtru adres, pod którym filtr umieszcza swoje dane. Po powrocie z funkcji Kombi zapisuje wypełniony przez filtr fragment pamięci na dysku. Funkcja zwraca rozmiar w bajtach obszaru, który zawiera dane filtru;
- setData jest wskaźnikiem do funkcji odczytującej dane (parametry) filtru z pliku. Jest ona zdefiniowana jako typedef void (XT_SetData)(RasterFilter* rf, void* p), gdzie:
 - rf jest wskaźnikiem do obiektu typu RasterFilter,
 - p jest wskaźnikiem do bloku pamięci.
Działanie tej funkcji polega na tym, że kiedy program Kombi odczytuje dokument z dysku, to dane odczytane z pliku za pomocą tej funkcji ustawiają odpowiednie parametry filtru;
- setPar jest wskaźnikiem do funkcji pozwalającej ustawiać dane w filtrze. Funkcję zdefiniowano jako

```
typedef void (XT_SetPar)(RasterFilter* rf, int type, LPARAM lParam), gdzie:
```

- rf jest wskaźnikiem do obiektu typu RasterFilter,
- type jest stałą ustalającą typ ustawianych danych. Zdefiniowane typy, to:

#define FR_DATA_TRANSPARENCY	0
//ustawia wartość przezroczystości,	
#define FR_DATA_CHECK	1
//ustawia opcje typu checkbox,	
#define FR_DATA_RADIO	2
//ustawia opcja typu radiobutton,	
#define FR_DATA_VAL0	100
//ustawia wartość parametru 1 (czyli suwaka //pierwszego),	
#define FR_DATA_VAL1	101
//ustawia wartość parametru 2,	
#define FR_DATA_VAL2	102
#define FR_DATA_VAL3	103
#define FR_DATA_VAL4	104
#define FR_DATA_VAL5	105
#define FR_DATA_VAL6	106
#define FR_DATA_VAL7	107
//ustawia wartość parametru 8;	

- lParam zawiera adres do zmiennej zawierającej ustawiane dane. Dla trzech pierwszych typów jest to adres do zmiennej typu **long**, dla pozostałych – typu **float**;
- getPar jest wskaźnikiem do funkcji odczytującej dane z filtru. Zdefiniowano ją jako typedef LPARAM (XT_GetPar)(RasterFilter* rf, int type), gdzie:
 - rf jest wskaźnikiem do obiektu typu RasterFilter,
 - type jest typem odczytywanych danych (patrz wyżej).

Funkcja zwraca adres do zmiennej zawierającej żądane wartości. Typy zwracanego adresu są takie, jak opisano wyżej;

- par[8] jest ośmioelementową tablicą obiektów typu rasterFilterPar;
- indexName jest identyfikatorem napisu (stringu) w zasobach. Napis powinien zawierać nazwę filtru;
- nOptCheck jest liczbą opcji typu **checkbox**;
- indexOptCheck jest indeksem napisu opisującego grupę opcji **checkbox**. Opisy kolejnych opcji powinny mieć indeksy o kolejnych wartościach poczynając od indexOptCheck+1;
- nOptRadio liczba opcji typu **radiobutton**;
- indexOptRadio jest indeksem napisu opisującego grupę opcji **radiobutton**. Opisy kolejnych opcji powinny mieć indeksy o kolejnych wartościach poczynając od indexOptRadio+1;
- nPar liczba parametrów (suwaków);
- indexPar jest indeksem napisu opisującego pierwszy suwak. Opisy kolejnych suwaków powinny mieć indeksy o kolejnych wartościach poczynając od indexPar+1;
- nAddOption liczba dodatkowych funkcji;

- `indexAddOption` jest indeksem napisu opisującego grupę dodatkowych funkcji. Opisy kolejnych funkcji powinny mieć indeksy o kolejnych wartościach poczynając od `indexAddOption+1`.

Klasę `rasterFilterPar` zdefiniowano następująco:

```
class rasterFilterPar{
public:
    float minVal;
    float maxVal;
    float val;
    int isReal;
    rasterFilterPar(){
        minVal=0;
        maxVal=0;
        val=0; isReal=0;
    };
    rasterFilterPar(float _minVal, float _maxVal, float _val,
        int _isReal){
        minVal=_minVal;
        maxVal=_maxVal;
        val=_val;
        isReal=_isReal;
    };
};
```

gdzie:

- `minVal` definiuje minimalną wartość edytowaną za pomocą suwaka,
- `maxVal` definiuje maksymalną wartość edytowaną za pomocą suwaka,
- `val` przechowuje aktualną wartość na suwaku,
- `isReal` jest zmienną ustalającą sposób traktowania pozostałych danych. Jeśli `isReal` jest równe 1, to parametry są traktowane jako zmiennoprzecinkowe, w przeciwnym wypadku, jak o całkowite.

Prześledźmy jak wypełniamy opisane wyżej struktury na podstawie przykładowego filtru, który znajdziemy w pliku `katalog_programu\api\filtr`.

```
extern "C" rasterFilterData* CALLBACK GetFilterData(){
    if(!rfd){
        rfd=new rasterFilterData;
        //przydzielenie pamięci dla obiektu,
        rfd->filterFunction=Pixel;
        //funkcją przeliczającą bitmapę będzie funkcja
        //Pixel (patrz przykładowy plik),
        rfd->addOption=PixelAddOption;
        //procedurą obsługującą dodatkowe funkcje
        //obsługiwane przez filtr będzie procedura
        //PixelAddOption,
        rfd->setData=Filter_SetDataColor;
        //Funkcja ustawiająca dane,
        rfd->getData=Filter_GetDataColor;
        //Funkcja pobierająca dane,
        rfd->setPar=0;
        //Pole zostanie wypełnione przez
        //program Kombi,
```

```
rfd->getPar=0;
//Pole zostanie wypełnione przez program
//Kombi,
rfd->par[0]=rasterFilterPar(0, 200, 100, 0);
//Parametr całkowity, wartość minimalna jest
//równa 0, wartość maksymalna – 200,
//wartość początkowa – 100,
rfd->par[1]=rasterFilterPar(0, 200, 100, 0); //jw.,
rfd->par[2]=rasterFilterPar(0, 100, 50, 0);
rfd->indexName=IDC_FILTER_NAME;
//indeks stringu zawierającego nazwę filtru,
rfd->nOptCheck=2;
//będą dwie opcje typu checkbox,
rfd->indexOptCheck=IDC_FILTER_OPTION;
//indeks stringu opisującego nazwy opcji
//typu checkbox,
rfd->nOptRadio=2;
//będą dwie opcje typu radiobutton,
rfd->indexOptRadio=IDC_FILTER_RADIO_OPTION;
//indeks stringu opisującego nazwy opcji
//typu radiobutton
rfd->nPar=3; //będą trzy suwaki,
rfd->indexPar=IDC_FILTER_PARAM1;
//indeks stringu opisującego pierwszy suwak,
rfd->nAddOption=2;
//będą dwie dodatkowe funkcje,
rfd->indexAddOption=IDC_FILTER_FUNCTION;
//indeks stringu opisującego nazwy
//dodatkowych funkcji,
    }
    return rfd;
}
```

Stałe zdefiniowano następująco:

```
#define IDC_FILTER_NAME 10100
#define IDC_FILTER_PARAM1 10101
#define IDC_FILTER_PARAM2 10102
#define IDC_FILTER_PARAM3 10103
#define IDC_FILTER_OPTION 10110
#define IDC_FILTER_OPTION1 10111
#define IDC_FILTER_OPTION2 10112
#define IDC_FILTER_RADIO_OPTION 10120
#define IDC_FILTER_RADIO_OPTION1 10121
#define IDC_FILTER_RADIO_OPTION2 10122
#define IDC_FILTER_FUNCTION 10130
#define IDC_FILTER_FUNCTION1 10131
#define IDC_FILTER_FUNCTION2 10132
```

Stałym tym odpowiadają następujące napisy zdefiniowane w pliku `filtr.rc`:

```
IDC_FILTER_NAME, "Przykładowy filtr"
```

```
IDC_FILTER_PARAM1, "Jasność (%):"
IDC_FILTER_PARAM2, "Kontrast (%):"
IDC_FILTER_PARAM3, "Podkolorowanie (%):"
```

```
IDC_FILTER_OPTION, "Opcje dodatkowe:"
IDC_FILTER_OPTION1, "Negatyw"
IDC_FILTER_OPTION2, "Podkolorowanie"
```

```
IDC_FILTER_RADIO_OPTION, "Podtyp:"
IDC_FILTER_RADIO_OPTION1, "Całość"
IDC_FILTER_RADIO_OPTION2, "Co drugi piksel"
```

```
IDC_FILTER_FUNCTION, "Funkcje dodatkowe:"
IDC_FILTER_FUNCTION1, "Stan neutralny"
IDC_FILTER_FUNCTION2, "Kolor podkolorowania"
```

Zwróćmy uwagę np. na pole `rfid->indexOptCheck`. Po-
lu temu przypisano wartość `IDC_FILTER_OPTION`, któ-
ra wynosi 10120. Jednocześnie zauważmy, że napis
o identyfikatorze `IDC_FILTER_OPTION` jest **Opcje do-
datkowe** i napis ten widnieje nad grupą opcji typu
checkbox na **rys. 2**. Kolejna wartością po wartości
10120 jest 10121 i wartości tej odpowiada identyfikator
`IDC_FILTER_RADIO_OPTION1`, któremu z kolei przypisano
napis **Negatyw**. Napis ten widnieje jako opis
pierwszej opcji w oknie pokazanym na **rys. 2**. Myślę,
że dalsze objaśnienia organizacji danych w filtrach nie
są już potrzebne.

Funkcja `extern "C" rasterFilterData* CALLBACK Get-
FilterData()` tworzy nagłówek i wypełnia go tak, jak opi-
sałem wyżej. Funkcja wywoływana jest przez Kombi
w momencie otwierania okna dialogowego filtrów ra-
strowych. Funkcja jest deklarowana jako `extern "C"`
ponieważ jest funkcją eksportową `dll-a` i musi być
wyszczególniona w pliku `def` (w sekcji `EXPORTS`):

```
EXPORTS
    GetFilterData      @2
    InitXtension       @1
```

Funkcja `InitXtension` zdefiniowana jako `extern "C"`
`KombiXtensionHeader* CALLBACK InitXtension(char* _path,
char* _KombiCopyNum)` jest wywoływana w momencie
instalowania rozszerzenia jak to opisano w rozdziale
poprzednim.

Funkcją, która jest „jądrem” filtru jest oczywiście
opisana wcześniej funkcja `Pixel`, którą przypisujemy
do pola `filterFunction`. To w tej funkcji następuje prze-
liczenie danych w bitmapie i w zasadzie tę funkcję
musi osoba pisząca nowy filtr rastrowy zaprogramować
od nowa. Pozostałe funkcje są zawarte w plikach
przykładowych i wystarczy je nieco dostosować do
własnych potrzeb. Istotnym elementem funkcji
`Pixel` są dwie pętle `for`. Pętla pierwsza (zewnętrzna)
przebiega wszystkie wiersze bitmapy, pętla

druga (wewnętrzna) – kolumny. Piksele w bitmapie
są ustawione trójkami lub czwórkami (w zależności
od przestrzeni kolorów). Jeśli mamy do czynienia
z bitmapą w trybie RGB (wartość `bps` będzie wtedy
równa 24), kolejne piksele w pamięci oznaczają na-
sycenie składowej B, G, R (to nie pomyłka, w syste-
mie Windows kolejność bajtów R i B jest zamienio-
na). W bitmapie typu CMYK, kolejne bajty oznaczają
C, M, Y, K. Aby dostać się do dowolnego piksela
w bitmapie trzeba więc wykonać następujący kod:

- dla bitmap RGB:


```
BYTE B=bits[wiersz*bpl+3*kolumna];
BYTE G=bits[wiersz*bpl+3*kolumna+1];
BYTE R=bits[wiersz*bpl+3*kolumna+2];
```

 gdzie:
 - `bits` jest adresem początku danych w bitmapie,
 - `wiersz` jest numerem wiersza licząc od dołu (w sy-
stemie Windows kolejność wierszy jest odwró-
cona, wiersz o indeksie `o` jest pierwszym wiers-
zem od dołu bitmapy),
 - `bpl` jest liczbą bajtów na wiersz (ta liczba musi
być wielokrotnością liczby 4, stąd w przykładzie
znajdujemy funkcję `ScanBytes` obliczającą tę wiel-
kość),
 - `kolumna` jest numerem kolumny w bitmapie,
 - mnożnik 3 wynika z faktu, że każdy piksel jest
opisany trzema bajtami;
- dla bitmap CMYK:


```
BYTE C=bits[wiersz*bpl+4*kolumna];
BYTE M=bits[wiersz*bpl+4*kolumna+1];
BYTE Y=bits[wiersz*bpl+4*kolumna+2];
BYTE K=bits[wiersz*bpl+4*kolumna+3];
```

 – mnożnik 4 wynika z faktu, że każdy piksel jest
teraz opisany czterema bajtami.

Po otrzymaniu składowych R, G, B lub C, M, Y, K
należy dokonać jakichś przeliczeń na tych danych.
Np. dodanie do wszystkich składowych stałej warto-
ści (dodatniej lub ujemnej) spowoduje zmianę jas-
ności bitmapy. Ale – dodanie tej wartości w jednym
kanale – spowoduje „podbicie” (lub „osłabienie”) je-
dnej składowej. Tu oczywiście nie ma żadnych ogra-
niczeń i w zasadzie wszystko zależy od naszej wyob-
raźni i fantazji. Dla przykładu pokażę, jak wyglądało-
by zwiększenie nasycenia składowej K:

```
K=min(255, max(0, K+L));
– L jest wartością dodatnią lub ujemną.
```

Funkcje `min` i `max` zabezpieczają zmienną `K` przed
przekroczeniem dozwolonego zakresu, który wynosi
 $0 \div 255$.

Po przeliczeniu składowych i otrzymaniu ich now-
ych wartości należy je wstawić z powrotem do bit-
mapy za pomocą kodu:

- dla bitmap RGB:


```
bits[wiersz*bpl+3*kolumna]=B;
bits[wiersz*bpl+3*kolumna+1]=G;
bits[wiersz*bpl+3*kolumna+2]=R;
```

- dla bitmap CMYK:

```
bits[wiersz*bpl+4*kolumna]=C;
bits[wiersz*bpl+4*kolumna+1]=M;
bits[wiersz*bpl+4*kolumna+2]=Y;
bits[wiersz*bpl+4*kolumna+3]=K;
```

Wewnątrz pętli zewnętrznej możemy wysłać do okna o uchwycie przekazany do funkcji Pixel komunikat PBM_SETPOS, w którym wartość wParam oznacza wysunięcie paska postępu w zakresie od 0 do 100.

2.7. Filtry wejścia/wyjścia

Filtry wejścia/wyjścia zbudowane są również w oparciu o rozszerzenie typu kxm. W tym wypadku nagłówek KombiXtensionHeader zastąpiony jest nagłówkiem KombiXtensionHeaderIOF zdefiniowany jako:

```
class KombiXtensionHeaderIOF:
public KombiXtensionHeader{
public:
int nFilter;
IOFilterData** iof;
~KombiXtensionHeaderIOF();
};
```

Jak widać, do zwykłego nagłówka KombiXtensionHeader dochodzą dwa pola:

- nFilter jest liczbą filtrów obsługiwanych przez dane rozszerzenie,
- iof jest wskaźnikiem do tablicy wskaźników o nFilter elementach. Tablica przechowuje wskaźniki do obiektów typu IOFilterData, które zdefiniowano następująco:

```
class IOFilterData{
public:
int type;
int flags;
ATOM fileType;
ATOM fileExt;
XT_IFilterFunction* iFunction;
XT_OFilterFunction* oFunction;
IOFilterData(char* fileType, char* fileExt,
int type, int flags);
~IOFilterData();
};
```

gdzie:

- type określa typ filtru. Dostępne są następujące typy:

```
#define IOFD_TYPE_RASTER 0
//filtr obsługuje bitmapy,
#define IOFD_TYPE_FONT 1
//filtr obsługuje kroje,
#define IOFD_TYPE_TEXT 2
//filtr obsługuje tekst,
#define IOFD_TYPE_META 3
//filtr obsługuje metapliki;
```

- flags zawiera dodatkowe flagi:

```
#define IOFD_FLAGS_IMPORT 1
//flaga oznacza, że filtr obsługuje import,
#define IOFD_FLAGS_EXPORT 2
//flaga oznacza, że filtr obsługuje eksport,
#define IOFD_FLAGS_HANDLE 4
//flaga oznacza, że w przypadku filtrów
//obsługujących bitmapy do komunikacji
//wykorzystany będzie blok pamięci,
//typu HGLOBAL przydzielony funkcją,
//GlobalAlloc, jeśli flags nie zawiera tej flagi,
//do komunikacji wykorzystywany jest
//obiekt typu TDib (z biblioteki OWL
//Borlanda);
```

- fileType jest atomem zawierającym nazwę formatu obsługiwanego przez filtr (np. w przykładowym filtrze jest to nazwa **Portable Network Graphics**);

- fileExt jest atomem zawierającym nazwę rozszerzenia obsługiwanego przez filtr importowo/eksportowy (np. **png**);

- iFunction jest funkcją importującą filtru i zdefiniowana jest jako typedef void* (XT_IFilterFunction)(HWND hwnd, char* filename, int option);

- hwnd jest uchwycem okna, do którego podpięte powinny być ewentualne komunikaty,

- filename jest ścieżką do pliku, który będzie wczytany przez filtr,

- option dodatkowe opcje importowe, aktualnie zdefiniowane są dwie flagi:

```
#define XT_IFILTER_OPTION_NOMESSAGE 1
//oznacza, że filtr importowy nie powinien
//wyświetlać ewentualnych komunikatów
//o błędach (np. generowany jest podgląd
//w Eksploratorze Kombi),
#define
```

```
XT_IFILTER_OPTION_NOCHECKLICENCE 2
//oznacza, że filtr nie powinien sprawdzać
//numeru licencji (np. mamy do czynienia
//z wersją demo, która i tak ma zablokowany
//zapis),
```

- funkcja powinna zwrócić uchwyt do pamięci lub wskaźnik do danych odpowiedniego typu w zależności od typu filtru. Jeśli filtr jest typu IOFD_TYPE_RASTER, to funkcja zwraca uchwyt do obszaru pamięci przydzielonego funkcją GlobalAlloc (jeśli wyszczególniona jest flaga IOFD_FLAGS_HANDLE) lub wskaźnik do obiektu typu TDib, jeśli wymieniona wyżej flaga nie jest wyszczególniona. Jeśli filtr jest typu IOFD_TYPE_FONT, to funkcja powinna zwrócić wskaźnik do obiektu typu NFont. Jeśli filtr jest typu IOFD_TYPE_TEXT, funkcja winna zapisać w katalogu tymczasowym plik w formacie *.kmt (KombiKora) i zwrócić adres do napisu zawierającego nazwę tego

pliku. Dla filtru typu IOFD_TYPE_META filtr powinien zapisać w katalogu tymczasowym ramkę w formacie Kombi (kmf) i zwrócić adres napisu zawierającego nazwę pliku tymczasowego.

Budowy obiektów NFont, pliku kmt oraz pliku kmf, ze względu na objętość i stopień złożoności nie opisuje w tej dokumentacji. Osoby zainteresowane podjęciem ewentualnej współpracy w tym zakresie proszę o nawiązanie korespondencji.

Jeśli funkcja nie może zaimportować pliku (uszkodzony plik lub nieobsługiwany format), powinna zwrócić wartość zero. Jeśli w czasie wykonywania funkcji wyświetlane jest dodatkowe okno dialogowe i użytkownik ewentualnie wycofa się z importu, funkcja powinna zwrócić wartość (void*) 0xFFFFFFFF;

- oFunction jest funkcją eksportującą zdefiniowaną jako typedef BOOL (XT_OFilterFunction)(HWND hwnd, void* data, char* filename, int option), gdzie:
 - hwnd jest uchwytem okna, do którego powinny być podjęte ewentualne komunikaty,
 - data jest uchwytem lub wskaźnikiem do eksportowanych danych. Aktualnie obsługiwany jest wyłącznie typ IOFD_TYPE_RASTER i dla niego data jest albo uchwytem do pamięci przydzielonej funkcją GlobalAlloc(), albo wskaźnikiem do obiektu typu TDiB (w zależności od tego, czy flaga IOFD_FLAGS_HANDLE jest wyszczególniona),
 - filename jest nazwą pliku, pod którą powinien być zapisany eksportowany plik,
 - option zależy od typu eksportowanych danych. Dla IOFD_TYPE_RASTER – option jest współczynnikiem kompresji w zakresie 0 ÷ 100;

Funkcja powinna zwrócić wartość TRUE, jeśli plik udało się zapisać bądź FALSE, jeśli nie.

Przykładowy filtr wejścia/wyjścia znajdziemy w katalogu `katalog_programu\api\io`. Filtr został przygotowany tak, jakby miał zaimportować lub wyeksportować plik w formacie PNG. Osoby zainteresowane tematem mogą spróbować w odpowiednie miejsca wstawić procedury kompresji i dekompresji plików tego typu (lub innego typu i zmienić przy tym odpowiednie opisy) tworząc w ten sposób w pełni funkcjonalne filtry importowo/eksportowe.

2.8. Komunikacja z programem głównym rozszerzeń typu exe

Przykład takiego rozszerzenia znajdziemy w katalogu `katalog_programu\api\cmd_list`. Jest to pro-

gram, który tworzy drzewo, do którego wstawia komendy obsługiwane przez Kombi. Dwukrotne kliknięcie w nazwę komendy w drzewie spowoduje wykonanie jej w Kombi.

Zwróćmy uwagę na sposób komunikowania się rozszerzenia z programem głównym. Nawiązanie dialogu między programami polega na wykonaniu następującego fragmentu kodu:

```
HWND hKombi=FindWindow("3N_KOMBI_SYSTEM", 0);
if(hKombi){
    //tu jest wykonywany kod w przypadku
    //nawiązania połączenia
}
```

3N_KOMBI_SYSTEM jest nazwą klasy głównego okna programu Kombi. Pomyślne nawiązanie połączenia spowoduje otrzymanie uchwyty hKombi. Mając ten uchwyt można wysłać do programu komunikat IDM_KOMBI_DATA, zdefiniowany jako:

```
#define IDM_KOMBI_DATA WM_USER+200
```

Parametr wParam w tym komunikacie przyjmuje jedną z opisanych niżej stałych, natomiast lParam oraz zwracana wartość zależy od wParam. Dalej omówię dostępne stałe wParam, wymagane dla nich wartości lParam oraz zwracane przez komunikat IDM_KOMBI_DATA wartości:

- #define IDM_KD_GETICON 0
 - lParam jest indeksem ikony,
 - funkcja zwraca uchwyt ikony o podanym indeksie;
- #define IDM_KD_GETBITMAP 1
 - lParam jest indeksem bitmapy odpowiadającym poleceniu o tym samym identyfikatorze,
 - funkcja zwraca atom, który zawiera nazwę pliku, w którym znajduje się żądana bitmapa. Bitmapa jest zapisana w formacie DIB;
- #define IDM_KD_GETSTRING 2
 - lParam jest indeksem komendy,
 - funkcja zwraca atom zawierający opis komendy w postaci stringu komentarz|krótki opis;
- #define IDM_KD_GETKEYNAME 3
 - lParam jest indeksem komendy,
 - funkcja zwraca atom, który zawiera opis skrótu klawiszowego przypisanego danej komendzie;
- #define IDM_KD_GETFILENAME 4
 - aktualnie nie obsługiwane;
- #define IDM_KD_GETCOMMAND 5
 - lParam jest liczbą z zakresu 1 ÷ 32767,
 - jeśli funkcja zwróci liczę większą od zera – żądana wartość lParam jest komendą;
- #define IDM_KD_EXECCOMMAND 6
 - lParam jest komendą,
 - jeśli komenda została wykonana zwracana jest wartość większa od zera, jeśli nie – wartość zero; W przypadku wszystkich wymienionych wyżej komunikatów indeks komendy przypisany interesującej nas funkcji można sprawdzić

- w Eksploratorze komend (indeksem jest nazwa pliku wyświetlona w pasku stanu Eksploratora);
- #define IDM_KD_LOADDOC 7
 - IParam jest atomem zawierającym nazwę dokumentu. Program otworzy ten dokument,
 - wartość zwracana nie ma znaczenia;
 - #define IDM_KD_MERGEDOC 8
 - IParam jest atomem zawierającym nazwę dokumentu. Program dołączy ten dokument do aktualnie otwartego,
 - wartość zwracana nie ma znaczenia;
 - #define IDM_KD_GETACTIVEFRAME 9
 - wartość IParam nie ma znaczenia,
 - funkcja zwraca typ aktywnej ramki; zdefiniowane są następujące typy:

#define FRAME_TEXT	0
//ramka z tekstem,	
#define FRAME_PASSER	1
//ramka z paserami,	
#define FRAME_LINE	2
//ramka z kształtem,	
#define FRAME_VECTOR	3
//ramka z grafiką wektorową,	
#define FRAME_RASTER	4
//ramka z grafiką rastrową,	
#define FRAME_PRINT	5
//ramka do drukowania,	
#define FRAME_GROUP	6
//grupa ramek,	
#define FRAME_METAFILE	7
//ramka z metaplikiem,	
#define FRAME_TABLE	9
//tabela,	
#define FRAME_VIRTUAL	10
//ramka wirtualna;	
- Uwaga! Jeśli aktywna jest grupa ramek, ale grupa ta składa się wyłącznie z ramek wektorowych zwracana jest wartość FRAME_VECTOR. Jeśli w dokumencie na aktywnej stronie nie ma jednej aktywnej ramki, funkcja zwraca wartość 0xFFFFFFFF;
- #define IDM_KD_GETRASTER 10
 - IParam nie ma znaczenia,
 - funkcja zwraca atom zawierający nazwę pliku tymczasowego, w którym znajduje się bitmapa zawarta w aktywnej ramce. Bitmapa zapisana jest w formacie DIB;
 - #define IDM_KD_GETVECTOR 11
 - IParam nie ma znaczenia,
 - funkcja zwraca atom zawierający nazwę pliku tymczasowego, w którym zapisana jest zawartość aktywnej ramki wektorowej. Plik jest zapisany w formacie EPS.

- #define IDM_KD_GETNET 12
 - IParam nie ma znaczenia,
 - funkcja zwraca atom zawierający nazwę pliku tymczasowego zawierającego dane projektora 3D. Dane są zapisane we własnym formacie binarnym;
- #define IDM_KD_SETRASTER 13
 - IParam jest atomem zawierającym nazwę pliku tymczasowego, w którym zapisana jest bitmapa w formacie DIB;
 - wartość zwracana nie ma znaczenia;
- #define IDM_KD_SETVECTOR 14
 - IParam jest atomem zawierającym nazwę pliku tymczasowego, w którym zapisane są dane wektorowe w formacie EPS;
 - wartość zwracana nie ma znaczenia;
- #define IDM_KD_SETMETAFILE 15
 - IParam jest atomem zawierającym nazwę pliku tymczasowego, w którym zapisany jest metaplik w formacie EMF;
 - wartość zwracana nie ma znaczenia;
- #define IDM_KD_SETNET 16
 - IParam jest atomem zawierającym nazwę pliku tymczasowego, w którym zapisane są dane projektora 3D we własnym formacie binarnym;
 - wartość zwracana nie ma znaczenia;
- #define IDM_KD_GET_ORIG_DIM 17
 - jeśli IParam jest równe 0, zwracana jest szerokość bitmapy, jeśli 1 – wysokość. Ten komunikat może być wysyłany nie do głównego okna aplikacji (hKombi), ale do okna filtrów rastrowych, czyli możemy go stosować podczas tworzenia filtrów w celu odczytania rzeczywistych wymiarów bitmapy. Parametry w i h przesyłane do funkcji Pixel omówionej w rozdziale poświęconym filtrom rastrowym są wymiarami bitmapy przetwarzanej, czyli w przypadku tworzenia podglądu, są to wymiary bitmapy w oknie podglądu. Komunikat tu omawiany zwróci natomiast wymiary rzeczywistej bitmapy;
- #define IDM_KD_GETCMDPATH 18
 - IParam nie ma znaczenia,
 - funkcja zwraca atom zawierający ścieżkę dostępu do komend (czyli katalog programu\config\komendy).

Wymienione wyżej komunikaty zostały zdefiniowane podczas dotychczasowych prac nad pakietem. Nie wyczerpują one oczywiście wszystkich potrzeb i możliwości. Jeśli osoby zainteresowane tworzeniem rozszerzeń będą potrzebowały innych danych, proszę o nawiązanie korespondencji w celu rozszerzenia oferowanego interfejsu.



Polecam zapoznanie się z przykładowymi kodami źródłowymi rozszerzeń, które można pobrać z naszego serwera (www.3n.com.pl/ftp/api/kombi_8_api_.exe).

3. Indeksy

3.1. Indeks rzeczowy

Filtry rastrowe.....	12
Filtry wejścia/wyjścia.....	17
Funkcja InitXtension.....	11
Funkcja XT_AddOption.....	14
Funkcja XT_FilterFunction.....	13
Funkcja XT_GetData.....	14
Funkcja XT_SetData.....	14
Instalowanie rozszerzeń.....	9
Interfejs API Kombi.....	9
Licencjonowanie rozszerzeń.....	10
Plik Kombi_Xtension.ini	9
Plik typu *.khp.....	12
Przycisk Informacje dodatkowe	12
Rozszerzenia typu exe.....	11
Rozszerzenia typu kxm.....	10
Rozszerzenia zewnętrzne.....	9
Struktura IOFilterData.....	17
Struktura KombiXtensionHeader.....	10
Struktura KombiXtensionHeaderIOF.....	17
Struktura RasterFilterData.....	13
Struktura rasterFilterPar.....	15

