Snabbguide för

Visma Integration för Visma Administration 500/1000/2000
Visma Integration för Visma Förening

Utvecklarpaket

Version: 5.1
December 2010

Beskrivning av Visma Integration för Visma Administration 500/1000/2000 och Visma Förening

Inledning

Visma Spcs är en av Sveriges största leverantörer av ekonomiprogram till mindre företag i Sverige. I Visma Spcs produktsortiment ingår många produkter, bl.a. Visma Administration, Visma Lön och Visma Skatt. De flesta programmen har tiotusentals användare och för att låta dessa slutanvändare koppla på för- och eftersystem till **Visma Administration 500/1000/2000** och **Visma Förening**, har Visma Spcs utvecklat ett generellt programmeringsgränssnitt API (Application Program Interface) och tagit fram produkterna **Visma Integration för Visma Administration 500/1000/2000** och **Visma Integration för Visma Förening**.

Möjlighet till integration

Genom **Visma Integration** är det alltså möjligt att integrera **Visma Administration 500/1000/2000** och **Visma Förening** med en extern programvara. För att kunna skapa denna integration krävs en programmeringsinsats. Visma Spcs har därför även tagit fram en utvecklardokumentation som innehåller all nödvändig information för programmering mot dessa program. Genom att teckna ett licensavtal med Visma Spcs får du som utvecklare tillgång till denna dokumentation och kan därmed skapa integrerade lösningar.

Om du har behov av att integrera en extern programvara med **Visma Administration 500/1000/2000** och **Visma Förening**, men inte själv har programmeringsresurser, ber vi dig besöka vår hemsida www.vismaspcs.se. Under rubriken ***Produkter och tjänster***, ***Fler produkter*** och **Visma *Integration*** finns en lista över befintliga API-leverantörer.

Vilken typ av lösning behöver jag?

Utvecklarpaketet för **Visma Integration** vänder sig i huvudsak till två olika kundgrupper:

1. Företag som är i behov av en integrerad lösning för eget bruk och som inte har för avsikt att sälja denna vidare till andra företag.

2. Kommersiella utvecklare som vill skapa integrerade lösningar med syftet att sälja dessa vidare till andra företag, dvs. till slutanvändare.

Integration för eget bruk

Om du själv har programmeringsresurser eller har möjlighet att köpa denna kunskap från något annat företag, kan du komplettera programmen **Visma Integration** med den nödvändiga utvecklardokumentationen. Detta alternativ innebär att du som slutkund själv köper både programmet och utvecklardokumentationen samt tecknar ett avtal med Visma Spcs för utveckling för eget bruk. En begränsning i detta avtal är att det inte ger dig rättighet att sälja den integrerade lösningen vidare till andra företag, utan kan endast använda den internt på ditt företag.

I denna lösning ingår:

- Programmet **Visma Integration för Visma Administration 500,1000** eller **2000,** eller **Visma Integration för Visma Förening**
- Licensavtal avseende **Visma Integration för Visma Administration 500, 1000** eller **2000,** eller **Visma Förening.**
- Rättighet att programmera mot API:et för eget bruk

Därutöver får du även tillgång till den nödvändiga utvecklardokumentationen som består av:

- Referensmanual med utförliga kodexempel i C++
- Exempelprojekt:
  - C++
  - C#
- Alla nödvändiga filer för programmering
- Rätt till en timmes fri handledning.
- Möjlighet till ytterligare handledning mot debitering f.n. 1 200 kr per timme exkl. moms. Debitering sker per påbörjad halvtimme.

Observera att en grundförutsättning för att kunna köpa någon av produkterna **Visma Integration för Visma Administration 500, 1000, 2000** respektive **Visma Integration för Visma Förening**, är att du har någon av produkterna **Visma Administration 500/1000/2000** eller **Visma Förening** samt ett giltigt serviceavtal för aktuellt program.

För ytterligare information hänvisas till Visma Spcs på telefon 0470 – 70 60 00 eller e-postadressen sales@vismaspcs.se.

Integration för kommersiella utvecklare

Om du har för avsikt att skapa en integrerad miljö och sälja den kompletta lösningen vidare till slutkunder, har du möjlighet att teckna ett utvecklaravtal med Visma Spcs.

I utvecklaravtalet ingår programmen **Visma Integration för Visma Administration 500/1000/2000** respektive **Visma Integration för Visma Förening** för utvecklarändamål, ej för eget bruk, samt nödvändiga filer för programmering. Utvecklaravtalet ger dig därmed möjlighet att skapa lösningar som kan integreras med både **Visma Administration 500, Visma Administration 1000, Visma Administration 2000** och **Visma Förening.**

Därutöver ingår i utvecklaravtalet:
- Referensmanual med utförliga kodexempel i C++
- Exempelprojekt:
  - C++
  - C#
- Alla nödvändiga filer för programmering
- Rätten att använda logotypen ”Licenserad API-utvecklare” vid kommunikation med slutkund.
- Rätt till en timmes fri handledning.
- Möjlighet till ytterligare handledning mot debitering. Debitering sker per påbörjad halvtimme.

För ytterligare information hänvisas till Visma Spcs på telefon 0470 – 70 60 00 eller e-postadressen sales@vismaspcs.se.

Kort teknisk beskrivning

Programmeringsspråk

API:et är framtaget för programmering i C++, men det finns även möjlighet att använda andra programmeringsspråk. Om du önskar information om programmering i andra språk ber vi dig kontakta oss på telefon 0470 – 70 61 61 eller e-postadressen sales.teknik@vismaspcs.se.

Systemkrav

Systemkraven för en integrerad lösning kan delas upp i två delar:

- Systemkrav för den externa produkten
- Systemkrav för produkterna **Visma Administration 500/1000/2000** respektive **Visma Förening**.

Visma Administration 500/1000/2000 respektive **Visma Förening** kan användas på något av operativsystemen Windows 7/Vista/XP.

För att använda programmet rekommenderas Pentium III-processor eller bättre. För installationen krävs minst 500 MB ledigt på hårddisken och därutöver ytterligare utrymme under användandet.

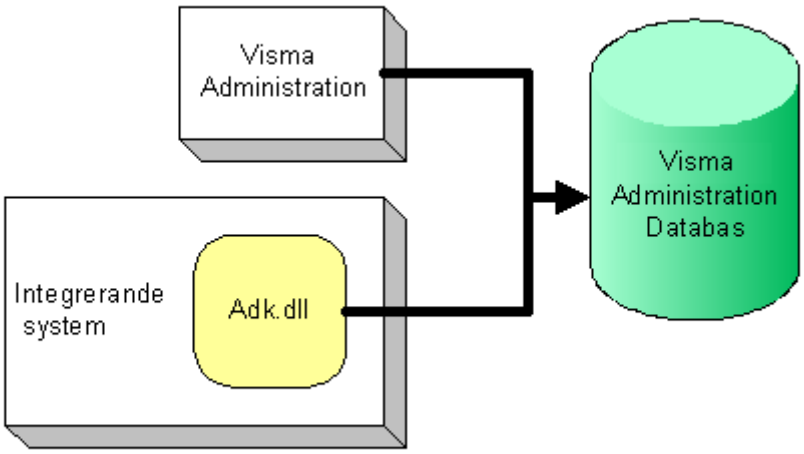
Du kan använda ditt externa program samtidigt som du använder **Visma Administration** respektive **Visma Förening.** Inga extra licenser är nödvändiga.

Vad är möjligt med hjälp av API:et?

API:et består av följande:

- En dll-fil, Adk.dll, som inkluderas i det integrerade systemet.
- En header-fil skriven i C, Adk.h, som exponerar ett antal funktioner vilka används för att hämta, lägga till eller ändra information i programmets databas.
- .Net-wrapper (installeras ihop med produkterna **Visma Administration 500/1000/2000** respektive **Visma Förening**)

Vid användandet av API:et kommer all kommunikation med programmets databas att ske via API:et.



API:et erbjuder möjligheten att arbeta med vissa förutbestämda delar av programmet. I vissa programdelar är det endast möjligt att läsa från databasen, alltså hämta första, hämta nästa och söka efter en post. Vilka programdelar som är möjliga att arbeta med samt vad som går att utföra i respektive del framgår av tabellen nedan.

Benämning		**Rättighet**
Artikel		Läsa, skriva och radera
Kund		Läsa, skriva och radera
Leverantör		Läsa, skriva och radera
Projekt	3)	Läsa, skriva och radera
Faktura		Läsa, skriva och radera rad
Leverantörsfaktura		Läsa, skriva och radera rad
Offert	1)	Läsa, skriva och radera rad
Order	2)	Läsa, skriva och radera rad
Manuell inleverans	2)	Läsa, skriva och radera
Manuell utleverans	2)	Läsa, skriva och radera
Beställning	1)	Läsa, Skriva och radera rad
Inventering	2)	Läsa och skriva
Artikelgrupp		Läsa, skriva och radera
Artikelkonto		Läsa
Betalsätt		Läsa, skriva och radera
Betalningsvillkor		Läsa, skriva och radera

Distrikt	2)	Läsa, skriva och radera
Enhet		Läsa, skriva och radera
Artikelbenämning		Läsa, skriva och radera
Paketartikel	1)	Läsa
Företagsuppgifter		Läsa och skriva
Inköpspriser		Läsa, skriva och radera
Konto		Läsa, skriva och radera
Kundkategori		Läsa, skriva och radera
Leveranssätt		Läsa, skriva och radera
Leveransvillkor		Läsa, skriva och radera
Prislista		Läsa, skriva och radera
Priser		Läsa, skriva och radera
Rabattavtal		Läsa, skriva och radera
Rabattavtalsrader		Läsa, skriva och radera
Resultatenhet		Läsa, skriva och radera
Språk	4)	Läsa, skriva och radera
Säljare	2)	Läsa, skriva och radera
Valuta	4)	Läsa, skriva och radera
Speditör	4)	Läsa, skriva och radera
Medlemmar	5)	Läsa, skriva och radera
Fria kategorier 1-10	5)	Läsa, skriva och radera
Inkommande följesedlar	1)	Läsa, skriva och radera
Kolli	2)	Läsa, skriva och radera
Leveransaviseringskolli	1)	Läsa
Avikande leveransadr.		Läsa, skriva och radera
Landskoder		Läsa, skriva och radera
Kundinbetalningar		Läsa, skriva och radera
Bortskrivningskoder		Läsa, skriva och radera
Företagsinställningar		Läsa
Utbetalningar		Läsa
Kontakter	4)	Läsa, skriva och radera
Verifikationer		Läsa och skriva
Bokföringsår		Läsa
Rabattkoder		Läsa och skriva
Kontakttitlar	4)	Läsa, skriva och radera
Kontaktgrupper	4)	Läsa, skriva och radera
Kontaktkopplingar	4)	Läsa, skriva och radera
Skattereduktion	3)	Läsa, skriva och radera
Avtal	4)	Läsa, skriva och radera rad
Skattereduktion på order	2)	Läsa, skriva och radera
Skattereduktion på avtal	2)	Läsa, skriva och radera
Verifikationsserier		Läsa
Behandlingshistorik		Läsa

1) Finns endast i Visma Administration 2000
2) Finns endast i Visma Administration 1000/2000
3) Finns endast i Visma Administration 1000/2000 och Visma Förening
4) Finns endast i Visma Administration 500/1000/2000
5) Finns endast i Visma Förening

Användning av API

Användning av API:et bygger på en grundstruktur där det finns ett antal grundfunktioner som är standardiserade och används till samtliga programdelar. Det går att dela upp de funktioner som finns i två kategorier. Den första kategorin är de funktioner som utför operationer mot databasen, t ex ”hämta första” eller ”lägga till”. Den andra kategorin är de funktioner som underlättar och gör användandet av API:et möjligt. Nedan följer en genomgång av hur API:et ska användas.

Funktioner som utför databasoperationer

Här beskrivs de funktioner som utför operationer mot databasen, t ex att lägga till en ny post eller söka upp en befintlig post. Centralt för dessa funktioner är en generell struktur som innehåller all information som kan hämtas eller skickas till **Visma Administration 500/1000/2000** och **Visma Förening**. Denna struktur innehåller olika fält beroende på vilken programdel som arbete ska ske mot. Samtliga fält som finns med i respektive struktur motsvarar ett fält i **Visma Administration 500/1000/2000** respektive **Visma Förening**.

AdkAdd

Funktionen `AdkAdd()` används för att addera något till databasen. Det första som måste göras är att skapa en datastruktur av den typ som ska användas. Detta görs med funktionen `AdkCreateData()`. I Adk.h-filen specificeras de delar av programmet som det är möjligt att arbeta med. Nedan följer ett exempel där en faktura med tillhörande rader skapas.

Först används funktionen `AdkCreateData()` för att skapa en datastruktur. Hit skickas `ADK_DB_INVOICE_HEAD` med som inargument. Inparametern i anropet talar om att strukturen ska användas för ett fakturahuvud. När strukturen för fakturahuvudet är skapad kan fälten i strukturen tilldelas olika värden, vilket görs med de olika `AdkSet`-funktionerna. Om t ex ett värde för kund-nummer ska anges på fakturan används Adk.h och där framgår att fältet `ADK_OOI_HEAD_CUSTOMER_NUMBER` i strukturen representerar kundnumret. Det framgår även att det är en sträng. Funktionen `AdkSetStr()` används för att tilldela detta fält ett värde. När valda fält är tilldelade i huvudet kan rader skapas och dessa kan kopplas till fakturahuvudet.

För att skapa rader anropas `AdkCreateDataRow()` med inparametrarna `ADK_DB_INVOICE_ROW` och antalet rader som ska skapas. Därefter skapas en vektor av strukturer för fakturarader, en för varje fakturarad. När aktuella fält är tilldelade i respektive rad ska de kopplas samman med huvudet. Kopplingen sker genom funktionen `AdkSetData()`. I strukturen för huvudet finns fältet `ADK_OOI_HEAD_ROWS` som utgör kopplingen till raderna.

När raderna är kopplade, ska antalet rader anges. Detta görs till fältet `ADK_OOI_HEAD_NROWS` i huvudet med hjälp av funktionen `AdkSetDouble()`. Initieringen av datastrukturen är därefter klar och funktionen `AdkAdd()` kan användas för att skriva till databasen. För att kunna avgöra vad det är som ska läggas till kontrollerar `AdkAdd()` först vilken programdel som strukturen representerar. Nedan följer ett komplett kodexempel i C++ på hur en faktura skapas:

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_INVOICE_HEAD);
error = AdkSetStr(pData, ADK_OOI_HEAD_CUSTOMER_NUMBER,"Kund1");
error = AdkSetStr(pData, ADK_OOI_HEAD_TYPE_OF_INVOICE, "F");

int nInvoiceRows = 3;
error = AdkSetDouble(pData, ADK_OOI_HEAD_NROWS, nInvoiceRows);
PADK_DATA pRowData = AdkCreateDataRow(ADK_DB_INVOICE_ROW, nInvoiceRows);

for(int i = 0; i < nInvoiceRows; i++)
{
   PADK_DATA pTempData = AdkGetDataRow(pRowData,i);
   error = AdkSetStr(pTempData, ADK_OOI_ROW_ARTICLE_NUMBER, "Artikel1");
   error = AdkSetDouble(pTempData, ADK_OOI_ROW_QUANTITY2, 10);
}
error = AdkSetData(pData, ADK_OOI_HEAD_ROWS, pRowData);
error = AdkAdd(pData);

AdkDeleteStruct(pData);
```

AdkSetSortOrder

Funktionen `AdkSetSortOrder()` bestämmer vilken sorteringsordning som ska användas till en databastabell. `AdkSetSortOrder()` tar två inparametrar, den första är en pekare till en datastruktur som är kopplad till den tabell som sorteringsordning ska bestämmas för och den andra används för att ange sorteringsordning. Alla sorteringsordningar finns specificerade i Adk.h-filen.

`AdkSetSortOrder()` används t ex vid en sökning. Sorteringsordningen gäller fram till dess att `AdkSetSortOrder()` anropas med en ny sorteringsordning för den aktuella tabellen.

AdkFind

Funktionen `AdkFind()` söker information i någon av de programdelar som API:et kan användas mot. På samma sätt som `AdkAdd()` är funktionen generell för samtliga delar av programmet. `AdkFind()` tar en inparameter, pekaren till den datastruktur sökning ska utföras i. Vilken sökordning som ska användas bestäms med hjälp av funktionen `AdkSetSortOrder()`, som beskrivits ovan. För att söka efter en kund kan sökning ske på bl.a. kundnummer eller kundnamn. Funktionen är uppbyggd så att det fält som sökning ska ske på tilldelas ett värde och sedan skickas datastrukturen in till funktionen. Övriga värden som det funna elementet innehåller fylls på i datastrukturen. Nedan följer ett kodexempel i C++ vilket sökning sker efter en kund (”Kund1”).

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_CUSTOMER);
ADK_SORT_ORDER nSortOrder = eCustomerNr;
error = AdkSetStr(pData, ADK_CUSTOMER_NUMBER, "Kund1");

error = AdkSetSortOrder(pData, nSortOrder);
error = AdkFind(pData);
```

Först skapas en datastruktur för typen kund. För att kunna söka på kundnummer måste därefter en variabel av typen `ADK_SORT_ORDER` skapas och variabeln måste ges värdet för kundnummer. För att kunna söka på kundnummer skapas därefter en variabel av typen `ADK_SORT_ORDER` och variabeln ges värdet för kundnummer. På nästa rad tilldelas det värde som sökning ska ske på, i detta fall ”Kund1”. Därefter anropas funktionen `AdkSetSortOrder()` för att ange sökordning och slutligen anropas `AdkFind()`.

Om sökningen går bra och en post hittas för det sökta värdet, fylls datastrukturen i med all data som denna post har. Om det finns fler poster med samma värde kommer den första att returneras. Sedan finns möjligheten att med hjälp av funktionen `AdkNext()` stega framåt. Om ingen träff sker på det sökta värdet returneras en felkod. Hur denna kod hanteras beskrivs i avsnittet om felhantering.

Nu innehåller datastrukturen som skickades in till `AdkFind()` alla de värden som den funna posten i databasen har. För att kunna utläsa värden från datastrukturen används `AdkGet`-funktionerna. Nedan följer ett kodexempel, som är en fortsättning på föregående exempel, där data hämtas från den funna posten.

```
char* sTemp = new char[51];
error = AdkGetStr(pData, ADK_CUSTOMER_NAME, &sTemp, 50);
int bTemp;
error = AdkGetBool(pData, ADK_CUSTOMER_REMINDER, &bTemp);
double dTemp;
error = AdkGetDouble(pData, ADK_CUSTOMER_CREDIT_LIMIT, &dTemp);
double dTemp2;
error = AdkGetDouble(pData, ADK_CUSTOMER_ROW_DISCOUNT, &dTemp2);
```

AdkFirst

Funktionen `AdkFirst()` hämtar den första posten i en programdel. Funktionen tar en inparameter, pekaren till den datastruktur där data från det första elementet ska skrivas.
Nedan följer ett kodexempel där den första artikeln som finns i databasen hämtas.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_ARTICLE);
error = AdkFirst(pData);
```

Här skapas en datastruktur av typen artikel, sedan anropas funktionen `AdkFirst()` och datastrukturen fylls med värden. För att läsa värden från datastrukturen används `AdkGet`-funktionerna, som beskrivits tidigare.

AdkLast

Funktionen `AdkLast()` hämtar den sista posten i en programdel. Funktionen tar en inparameter, pekaren till den datastruktur där data från det sista elementet ska skrivas.
Nedan följer ett kodexempel i C++ där den sista artikeln som finns i databasen hämtas.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_ARTICLE);
error = AdkLast(pData);
```

Här skapas en datastruktur av typen artikel, sedan anropas funktionen `AdkLast()` och datastrukturen fylls med värden. För att läsa värden från datastrukturen används `AdkGet`-funktionerna, som beskrivits tidigare.

AdkNext

Funktionen `AdkNext()` hämtar nästa post i en programdel. Förutsättningen för att kunna använda denna funktion är att en post tidigare har lästs in i datastrukturen, t ex med `AdkFind()` eller `AdkFirst()`. Nedan följer ett kodexempel i C++ på hur alla artiklar läses upp.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_ARTICLE);
error = AdkFirst(pData);
do
{
   char* temp = new char[31];
   error = AdkGetStr(pData, ADK_ARTICLE_NAME, &temp, 30);
   printf(”%s\n”, temp);
   error = AdkNext(pData);
   delete[] temp;
}
while(error.lRc != ADKE_EOF);
```

Här skapas en datastruktur av typen artikel. Först hämtas den första posten i artikelregistret och sedan skrivs dess artikelnummer på skärmen. Därefter anropas funktionen `AdkNext()`. Finns det ingen nästa post kommer ett felmeddelande att returneras och loopen bryts. För att läsa värden från datastrukturen används `AdkGet`-funktionerna, som beskrivs i avsnittet om `AdkFind()`.

AdkPrevious

Funktionen `AdkPrevious()` hämtar föregående post i en programdel. Förutsättningen för att kunna använda denna funktion är att en post tidigare har lästs in i datastrukturen, t ex med `AdkFind()` eller `AdkLast()`. Nedan följer ett kodexempel i C++ på hur alla artiklar läses upp.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_ARTICLE);
error = AdkLast(pData);
do
{
   char* temp = new char[31];
   error = AdkGetStr(pData, ADK_ARTICLE_NAME, &temp, 30);
   printf(”%s\n”, temp);
   error = AdkPrevious(pData);
```

```
   delete[] temp;
}
while(error.lRc != ADKE_BOF);
```

Här skapas en datastruktur av typen artikel. Först hämtas den sista posten i artikelregistret och sedan skrivs dess artikelnummer på skärmen. Därefter anropas funktionen `AdkPrevious()`. Finns det ingen nästa post kommer ett felmeddelande att returneras och loopen bryts. För att läsa värden från datastrukturen används `AdkGet`-funktionerna, som beskrivs i avsnittet om `AdkFind()`.

AdkUpdate

Funktionen `AdkUpdate()` uppdaterar en befintlig post i databasen. Användandet av funktionen sker på följande vis:

En datastruktur skapas med `AdkCreateData()` och tilldelas de värden som ska ändras. För att peka ut vilken post som ska ändras används det aktuella registrets nyckel, t ex artikelnummer om det är en artikelpost som ska uppdateras. Nedan följer ett kodexempel i C++ på hur en artikel uppdateras.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_ARTICLE);
error = AdkSetStr(pData, ADK_ARTICLE_NUMBER, "Artikel1");
error = AdkSetStr(pData, ADK_ARTICLE_NAME, "UppdateradArtikel1");
error = AdkSetStr(pData, ADK_ARTICLE_SHORT_NAME, "Uppd.Art1");

error = AdkUpdate(pData);
```

Först pekas den aktuella artikeln ut med hjälp av dess artikelnummer, sedan tilldelas de värden till aktuella fält som ska uppdateras. När värden är tilldelade anropas funktionen `AdkUpdate()` som utför uppdateringen i databasen. Det är inte möjligt att uppdatera nyckelfält.

AdkDelete

Funktionen `AdkDelete()` raderar en post i databasen. Innan `AdkDelete()` anropas måste en datastruktur skapas med `AdkCreateData()`. För att peka ut vilken post som ska raderas används det aktuella registrets nyckel, t ex artikelnummer när en artikelpost ska raderas. Nedan följer ett kodexempel i C++ på hur en artikel raderas.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));

PADK_DATA pData = AdkCreateData(ADK_DB_ARTICLE);
error = AdkSetStr(pData, ADK_ARTICLE_NUMBER, "Artikel1");
error = AdkDelete(pData);
```

Först pekas den aktuella artikeln ut med hjälp av dess artikelnummer och sedan anropas funktionen `AdkDelete()` som utför raderingen i databasen. Om utförandet av funktionen `AdkDelete()` gick bra kommer minnet som datastrukturen `pData` upptar att avallokeras.

AdkDeleteRow

Funktionen `AdkDeleteRow()` fungerar på liknande sätt som `AdkDelete()`. `AdkDeleteRow()` raderar en rad i en datastruktur. Funktionen används på följande sätt:

En datastruktur skapas med hjälp av `AdkCreateData()`. För att peka ut vilken post som ska raderas används det aktuella registrets nyckel, t ex ordernummer om det är en orderrad som ska raderas. För att peka ut vilken rad som ska raderas används radens radnummer, som skickas med som en inparametrar. Nedan följer ett kodexempel i C++ på hur en orderrad raderas.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));
```

```
PADK_DATA pData = AdkCreateData(ADK_DB_ORDER_HEAD);
error = AdkSetDouble(pData, ADK_OOI_HEAD_DOCUMENT_NUMBER, 2000023);
error = AdkDeleteRow(pData, 1);
```

Först pekas den aktuella ordern ut med hjälp av dess dokumentnummer och sedan anropas funktionen `AdkDeleteRow()` med pekaren till orderhuvudets datastruktur samt vilken orderrad som ska raderas. I fallet ovan är det rad nummer 1. Om utförandet av funktionen `AdkDeleteRow()` gick bra kommer datastrukturen `pData` att fyllas på med det aktuella orderhuvudets data. Även de kvarvarande raderna kommer att skapas och fyllas med aktuell data.

AdkGetCustomerPrice

Funktionen AdkGetCustomerPrice() returnerar en specifik kunds pris med avseende på de parametrar som anges.

```
AdkGetCustomerPrice(CHAR* strCustomer, CHAR* strArticle, DOUBLE dQuantity, BOOL bInvDisc, BOOL
bRowDisc, BOOL bInclVAT, INT iSelCir, DOUBLE* dVal)
```

- `strCustomer` – En charpekare som innehåller kundnumret till vilket man vill beräkna ett pris.
- `strArticle` – En charpekare som innehåller artikelnumret till vilket man vill få fram priset.
- `dQuantity` – Antalet artiklar som är tänkta att säljas (för att beräkna artikelns stafflingpris).
- `bInvDisc` – Om kundens eventuella fakturarabatt ska användas vid beräkningen eller inte.
- `bRowDisc` – Om kundens eventuella radrabatter ska tas med vid beräkningen eller inte.
- `bInclVat` – Om priset ska vara inklusive eller exklusive moms.
- `bInclVAT` – I vilket pris beräkningen ska ske
    - 0 – Kundens valuta
    - 1 – Inhemsk valuta
    - 2 – Slutgiltig prislistas valuta
- `dVal` – Minnesadress där värdet kommer att skrivas.

AdkGetVatprecentOnArticle

Funktionen användas för att hämta momssatsen för en specifik artikel i artikelregistret. Då det hittills varit en flerstegsprocess att få ut momssatsen skapades denna funktion för att underlätta hämtningen. Funktionen tar två inparametrar

```
AdkGetVatpercentOnArticle(CHAR* strArticle, DOUBLE* dVal)
```

- `strArticle` – En charpekare som innehåller det artikelnummer för vilken momssats önskas.
- `dVal` – Minnesadress där värdet kommer att skrivas.

AdkGetSie

Funktionen används för att generera SIE-filer på motsvarande sätt som går att göra i programmet.

Hjälpfunktioner som inte utför databasoperationer

Här beskrivs de funktioner som inte arbetar direkt mot databasen men som ändå behövs för att API:et ska fungera korrekt.

AdkOpen

Funktionen `AdkOpen()` används för att öppna en databas till ett befintligt företag. Funktionen tar två inparametrar, den första är en sträng som pekar ut var gemensamma filer för **Visma Administration**

500/1000/2000 respektive **Visma Förening** är installerat. Den andra är en sträng som pekar ut var det aktuella företaget är sparat.

AdkOpenEx

Funktionen `AdkOpenEx()` används för att öppna en databas till ett befintligt företag. Funktionen tar tre inparametrar, den första är en sträng som pekar ut var gemensamma filer för **Visma Administration 500/1000/2000** respektive **Visma Förening** är installerat. Den andra är en sträng som pekar ut var det aktuella företaget är sparat. Den tredje används för att slå av respektive på loggning av funktionsanrop via adk.dll.

AdkClose

Funktionen `AdkClose()` stänger den koppling mot databasen som API:et har öppnat.

AdkCreateData

Funktionen `AdkCreateData()` skapar en datastruktur och returnerar en pekare till den skapade datastrukturen. `AdkCreateData()` tar en inparameter, som anger vilken typ av struktur som ska skapas. Observera att strukturerna endast får skapas av `AdkCreateData()` eller `AdkCreateDataRow()` och tas bort med `AdkDeleteStruct()`.

AdkCreateDataRow

Funktionen `AdkCreateDataRow()` skapar en datastruktur och returnerar en pekare till den skapade datastrukturen. `AdkCreateDataRow()` tar två inparametrar, som anger vilken typ av och antalet strukturer som ska skapas.

AdkDeleteStruct

Funktionen `AdkDeleteStruct()` avallokerar minne till en skapad datastruktur. `AdkDeleteStruct()` tar en inparameter, pekaren till den datastruktur som ska avallokeras. Den returnerar en datastruktur av typen `ADK_ERROR`.

AdkResetStruct

Funktionen `AdkResetStruct()` nollställer minnet till en skapad datastruktur. `AdkResetStruct()` tar en inparameter, pekaren till den datastruktur som ska nollställas. Den returnerar en datastruktur av typen `ADK_ERROR`.

Felhantering

Felhanteringen i API:et använder sig av en datastruktur. Denna struktur är byggd på följande vis:

```
typedef struct _Error
{
   LONG lRc;
   LONG lDbTable;
   LONG lField;
   LONG lFunction;
   LONG lProgramPart;
} ADK_ERROR, *PADK_ERROR;
```

Datastrukturen består av 5 element, dessa beskrivs nedan:

- `lRc` – Felmeddelandet, alla kontroller bör göras mot detta värde. Det antar värdet 0 från början. Är det oförändrat har allt gått bra, är det förändrat har något gått fel.
- `lDbTable` - Pekar ut vilken databastabell som användes när felet uppstod.
- `lField` - Pekar ut vilket fält i aktuell databas som användes när felet uppstod.

- `lFunction` - Pekar ut vilken funktion som anropades innan felet uppstod, t ex ”AdkAdd”. Även det som ska läggas till pekas ut, t ex ”AdkAddCustomer”.
- `lProgramPart` - Pekar ut den del i API:et där felet uppstod (används internt).

När en funktion returnerar en datastruktur av typen `ADK_ERROR` ska en validering ske mot det returvärde som ges. Är värdet skilt från `ADKE_OK` har ett fel uppstått. Valideringen ska ske mot den kod som finns i `lRc`, vilket kan ske på följande sätt.

```
if(error.lRc != ADKE_OK)
{
   //Error!
}
```

Till alla felkoder som beskrivs ovan kan feltexter hämtas. Detta görs med hjälp av funktionen `AdkGetErrorText()`. Denna funktion tar fyra inparametrar:

```
AdkGetErrorText(ADK_ERROR* error, ADK_ERROR_TEXT_TYPE nErrorTextType,  CHAR** achBuf, INT
iLen)
```

Nedan följer en redogörelse för de fyra inparametrarna:

- `error` - Datastrukturen som innehåller felmeddelanden.
- `nErrorTextType` - Vilken typ av fel som ska hämtas. Dessa definieras i enumen `ADK_ERROR_TEXT_TYPE` som finns definierad i Adk.h.
- `achBuf` – Pekare till en sträng. Här kommer feltexten att skrivas.
- `iLen` - Integer som definierar hur många tecken som finns i den fjärde parametern.

AdkSetWarningFunction

Funktionen `AdkSetWarningFunction()` anger vilken funktion som ska anropas vid varningar. Hanteringen av varningar är i API:et löst genom att de rapporteras till en funktion som ligger utanför API:et och som måste skapas av den som utvecklar mot API:et. Funktionen ska ta en pekare till en datastruktur av typen `ADK_ERROR`. När en varning uppstår kommer denna funktion att anropas. `AdkSetWarningFunction()` anropas för att ange vilken funktion som ska användas för att ta emot varningar. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkSetWarningFunction(void(*function)(ADK_ERROR*))
```

- `function` – Funktionspekare till en funktion som tar en pekare till en datastruktur av typen `ADK_ERROR` som inparameter.

Datum

I databasen behandlas datum som datatypen long (Juliandatum). För att underlätta användningen tillhandahåller API:et två datumfunktioner. Den första är `AdkLongToDate()` som konverterar en long till ett vanligt datum i en sträng. Funktionen tar tre inparametrar där den första är det long värde som ska omvandlas, den andra en strängpekare där datumet skrivs och den tredje är längden på den sträng där datumet ska skrivas. Skickas ett felaktigt värde till den första parametern kommer funktionen att ge dagens datum.

Den andra funktionen är `AdkDateToLong()` som konverterar ett datum i form av en sträng till en long (Juliandatum). Funktionen tar två inparametrar där den första är en `charpekare` till strängen som ska konverteras och den andra är en referens till den long där datum ska skrivas som long (Juliandatum). Vid användandet av `AdkDateToLong()` är det viktigt att strängen är i formatet ”YYYY-MM-DD”, vid felaktig användning kommer dagens datum att anges.

`AdkLongToDate()` och `AdkDateToLong()` returnerar en datastruktur av typen `ADK_ERROR`.

Tidsstämplar

I databasen behandlas datum som datatypen long (__time32_t). För att tilldela eller läsa en tidsstämpel ska funktionerna `AdkSetDate()` respektive `AdkGetDate()` användas. För att underlätta hanteringen av tidsstämplar tillhandahåller API:et två funktioner för konvertering till och från __time32_t, `AdkLongToDateTime()` och `AdkDateTimeToLong()`.

För att göra sökningar i tidsstämpelordning, så sätter man sökordning till `eTimeStamp` och anger en starttidpunkt till fältet för tidsstämpeln. Poster som ändrats efter denna tidpunkt finns efter den post man eventuellt träffat. För att ta reda på vilket fält som är tidstämpel i aktuellt register använder man funktionen `AdkGetTimeStampField()`. Skulle registret sakna tidsstämpelfält returneras ett fel. Nedanstående kod är ett exempel på hur man kan göra med leverantörsfakturor.

```
ADK_ERROR error;
ZeroMemory(&error, sizeof(ADK_ERROR));
PADK_DATA pData = AdkCreateData(ADK_DB_SUPPLIER_INVOICE_HEAD);
if(pData==NULL)
{
    //Avbryt hanteringen
}

error= AdkSetSortOrder(eTimeStamp);
if(error.lRc != ADKE_OK)
{
    CErrorHandling rErrorHandling(&error);
}

int iTimeStampField;
error= AdkGetTimeStampField(pData, &iTimeStampField);
if(error.lRc != ADKE_OK)
{
    CErrorHandling rErrorHandling(&error);
}

Char szDateTime[] = "2010-10-01 12:00:00";
Long lDateTime;
error= AdkDateTimeToLong(szDateTime, &lDateTime);
if(error.lRc != ADKE_OK)
{
    CErrorHandling rErrorHandling(&error);
}

error= AdkSetDate(pData, iTimeStampField, lDateTime);
if(error.lRc != ADKE_OK)
{
    CErrorHandling rErrorHandling(&error);
}

for(error = AdkFind(pData); error == ADKE_OK; error = AdkNext(pData))
{
    double dLoepnr;
    error = AdkGetDouble(pData, ADK_SUP_INV_HEAD_GIVEN_NUMBER, &dLoepnr);
    if(error.lRc == ADKE_OK)
    {
        printf("%f\n", dLoepnr);
    }
}
```

AdkGet

`AdkGet` är ett antal funktioner som hämtar data ur en datastruktur och används t ex efter funktionen `AdkFind()` för att läsa de värden som blivit funna. Funktionerna tar tre inparametrar. Den första inparametern är pekaren till den datastruktur där data ska hämtas. Den andra inparametern är fältid till det fält där värdet ska hämtas.

Den tredje inparametern är en referensvariabel där det önskade värdet kommer att skrivas. `AdkGetStr()` tar även en fjärde inparameter som anger referensvariabelns längd. De `AdkGet`-funktioner som finns tillgängliga är:

- AdkGetStr()
- AdkGetBool()
- AdkGetData()
- AdkGetDouble()
- AdkGetDate()

AdkGet returnerar en datastruktur av typen ADK_ERROR.

AdkSet

AdkSet är ett antal funktioner som skriver värden i en datastruktur. De används t ex innan funktionen AdkAdd anropas (som skriver värdena från den aktuella datastrukturen till databasen). Samtliga AdkSet-funktionerna tar tre inparametrar. Den första inparametern är pekaren till den datastruktur där värden ska skrivas. Den andra inparametern är fältid till det fält där värdet ska skrivas. Den tredje inparametern är det värde som ska skrivas. De AdkSet–funktioner som finns tillgängliga är:

- AdkSetStr()
- AdkSetBool()
- AdkSetData()
- AdkSetDouble()
- AdkSetDate()

AdkSet returnerar en datastruktur av typen ADK_ERROR.

AdkGetType

Funktionen AdkGetType() används för att hämta ett fälts datatyp. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkGetType(ADK_DATA* pData, INT iFieldId, ADK_FIELD_TYPE* eType)
```

- pData – Pekaren till den datastruktur där datatypen ska hämtas.
- iFieldId - Fältid till det aktuella fältet.
- eType - Pekare till en variabel av typen ADK_FIELD_TYPE. Denna typ kan endast anta godkända datatypvärden. ADK_FIELD_TYPE finns definierad i Adk.h.

AdkGetType() returnerar en datastruktur av typen ADK_ERROR.

AdkGetLength

Funktionen AdkGetLength() hämtar ett fälts längd. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkGetLength(ADK_DATA* pData, INT iFieldId, INT* iLength);
```

- pData - Pekaren till den datastruktur där längden ska hämtas.
- iFieldId - Fältid till det aktuella fältet.
- iLength - Pekare till en integer där längden på fältet skrivs.

AdkGetLength() returnerar en datastruktur av typen ADK_ERROR.

AdkGetDecimals

Funktionen AdkGetDecimals() hämtar det antal decimaler ett fält kan innehålla. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkGetDecimals(ADK_DATA* pData, INT iFieldId, INT* iDec)
```

- `pData` – En pekare till den datastruktur där antalet decimaler ska hämtas från ett fält.
- `iFieldId` - Id till det fält där antalet decimaler ska hämtas.
- `iDec` – Minnesadressen till en `int`, här kommer antalet decimaler att anges.

`AdkGetDecimals ()` returnerar en datastruktur av typen `ADK_ERROR`.

AdkGetFieldName

Funktionen `AdkGetFieldName()` hämtar namnet på ett fält, det alias som finns till varje fält i databasen. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkGetFieldName(INT iDataBaseId, INT iFieldId, CHAR** ppsValue , INT iLen)
```

- `iDataBaseId` – Den databastabell där fältnamnet ska hämtas.
- `iFieldId` - Fältid till det aktuella fältet.
- `ppsValue` - Pekare till en charpekare där namnet på fältet skrivs.
- `iLen` – Längden på ppsValue.

`AdkGetFieldName()` returnerar en datastruktur av typen `ADK_ERROR`.

AdkGetDataRow

Funktionen `AdkGetDataRow()` returnerar en pekare till en rad i en datastruktur med flera element. Funktionen används t ex när en speciell rad ska pekas ut i en struktur med flera fakturarader. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkGetDataRow(ADK_DATA* pDataRow, INT iIx)
```

- `pDataRow` – Pekaren till datastrukturen.
- `iIx` – Indexet till önskad rad.

AdkIsReadWrite

Funktionen `AdkIsReadWrite()` returnerar en `boolean` som anger om ett fält är `ReadWrite` eller ej. Om returvärdet är sant innebär det att det aktuella fältet är `ReadWrite`**,** om returvärdet är falskt innebär det att det aktuella fältet är `ReadOnly`. Nedan följer en definition av funktionen och dess inparametrar:

```
AdkIsReadWrite(ADK_DATA* pData, INT iFieldId)
```

- `pData` – Pekaren till aktuell datastruktur.
- `iFieldId` – Fältid till det aktuella fältet.

AdkGetVariant

Information

Visma Integration för Visma Administration 500/1000/2000 och **Visma Integration för Visma Förening** använder alla filen adk.dll. Dessa program innehåller olika moduler och funktionalitet, tex finns offerthantering bara i **Visma Administration 2000**. Det kan därför vara nödvändigt att ta reda på vilket program man jobbar mot så att ens egna program kan anpassas till det program som finns installerat hos slutanvändaren. Funktionen ersätter `AdkGetAdmSize()` resp `AdkGetAdmSizeEx()`, vilka inte bör användas från version 5.0.

Funktionen `AdkGetVariant()` hämtar information ur adk.dll om vilket program som är installerat. Detta kan erhållas antingen som ett heltalsvärde eller som en textsträng, beroende på medskickade inparametrar.

- `INT* piVariant` – En `int`-pekare som är definierad utanför API:et. Värdet kommer att skrivas där. `NULL` tillåtet.

- `BOOL* pbApi` – En `int`-pekare som är definierad utanför API:et. Värdet kommer att skrivas där. `NULL` tillåtet.

- `CHAR** chBuf` – Minnesadressen till en `char`-pekare som är definierad utanför API:et. Värdet kommer att skrivas in där. `NULL` tillåtet. Minst 44 tecken bör reserveras för resultatet utanför API:et för att hela texten ska få plats.

- `INT iBufSize` – storleken på textbufferten som reserverats för resultatet. Om `chBuf` är NULL saknar denna parameter betydelse.

- `CHAR* pszSystemPath` – en sträng som pekar ut var **Visma Administration 500/1000/2000** respektive **Visma Förening** är installerade.

Om exempelvis **Visma Integration för Visma Administration 2000** är installerat skrivs till `piVariant` värdet ADKI_ADMIN2000 och texten `”Visma Integration för Visma Administration 2000”` skrivs till `chBuf`. Värdet `TRUE (1)` skrivs till `pbApi`.

Saknas licens för **Visma Integration** sätts `pbApi` till `FALSE (0)`.

Möjliga kombinationer är:

- `ADKI_ADMIN2000 - ”Visma Integration för Visma Administration 2000” - TRUE`
- `ADKI_ADMIN1000 - ”Visma Integration för Visma Administration 1000” - TRUE`
- `ADKI_ADMIN500 - ”Visma Integration för Visma Administration 500” - TRUE`
- `ADKI_FORENING - ”Visma Integration för Visma Förening” - TRUE`
- `ADKI_ADMIN2000 - ”Visma Administration 2000” - FALSE`
- `ADKI_ADMIN1000 - ”Visma Administration 1000” - FALSE`
- `ADKI_ADMIN500 - ”Visma Administration 500” - FALSE`
- `ADKI_ADMIN200 - ”Visma Administration 200” - FALSE`
- `ADKI_FORENING - ”Visma Förening” - FALSE`

Om licens finns för Visma Integration skrivs `TRUE` till `pbApi`

`AdkGetVariant()` returnerar en datastruktur av typen `ADK_ERROR`, vilken innehåller information om resultatet av funktionen. Om felkod erhålls skrivs inga värden till `piAdmSize` respektive `chBuf`.

Dynamisk meny

I **Visma Administration 500/1000/2000** respektive **Visma Förening** finns möjlighet att själv lägga till val i programmenyn. Detta görs med hjälp av filen *DynamicMenu.xml*. Denna fil används för Menyn och kommer endast att finnas tillgänglig för de kunder som har **Visma Integration för Visma Administration 500/1000/2000** eller **Visma Integration för Visma Förening**. Den kommer dessutom enbart visas om det finns något valbart alternativ.

Från och med version 5.1 kan man lägga sin egen kopia lokalt på datorn i samma mapp som programmets inifil finns. Om det finns en *DynamicMenu.xml* på denna plats så kommer denna väljas istället för den fil som ligger bland gemensamma filer. Dett möjliggör olika versioner av *DynamicMenu.xml* på olika datorer med olika versioner av operativsystemet, samt ev olika sökvägar till gemensamma filer resp företag.

Tre egenskaper finns för fria texter i Menu resp MenItem. Dessa används inte i dagsläget av programmet. De kan tex användas för att identifiera egna menyer vid uppdateringar och installationer. Dessa reserveras för framtiden och kommer vid användning underlätta när flera integrationer från en eller flera partners finns i den dynamiska menyn.

Följande tre taggar med tillhörande parametrar finns att tillgå

- <Menu> – Submeny
    - Name – Namnet på menyn
    - Company – Fri text
    - App – Fri text
    - Description – Fri text
- <MenuItem>
    - Path – Sökväg till programmet
    - Parameters – Parametrar som ska skickas med vid exekveringen
        - %P – Ger programnamnet
        - %F – Ger sökväg till aktivt företag
        - %G – Ger sökväg till gemensamma filer
        - %R – Ger aktuellt register
        - %U – Ger unikt id
        - Övrigt ger exakt det som skrivs
    - Company – Fri text
    - App – Fri text
    - Description – Fri text
- <Separator>

Exempel

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<DynamicMenuSetup>
<Menu Name="Notepad">
              <MenuItem Path="C:\NOTEPAD.EXE" Parameters="C:\Text.txt" Company="Visma
              Spcs">Anteckningar</MenuItem>
</Menu>
<Separator />
<MenuItem Path="C:\Testprogram.exe" Parameters="Test %F " Description="Detta är ett
testprogram">Testprogram</MenuItem>
</DynamicMenuSetup>
```