

Verwendung von User Function Libraries mit Crystal Reports

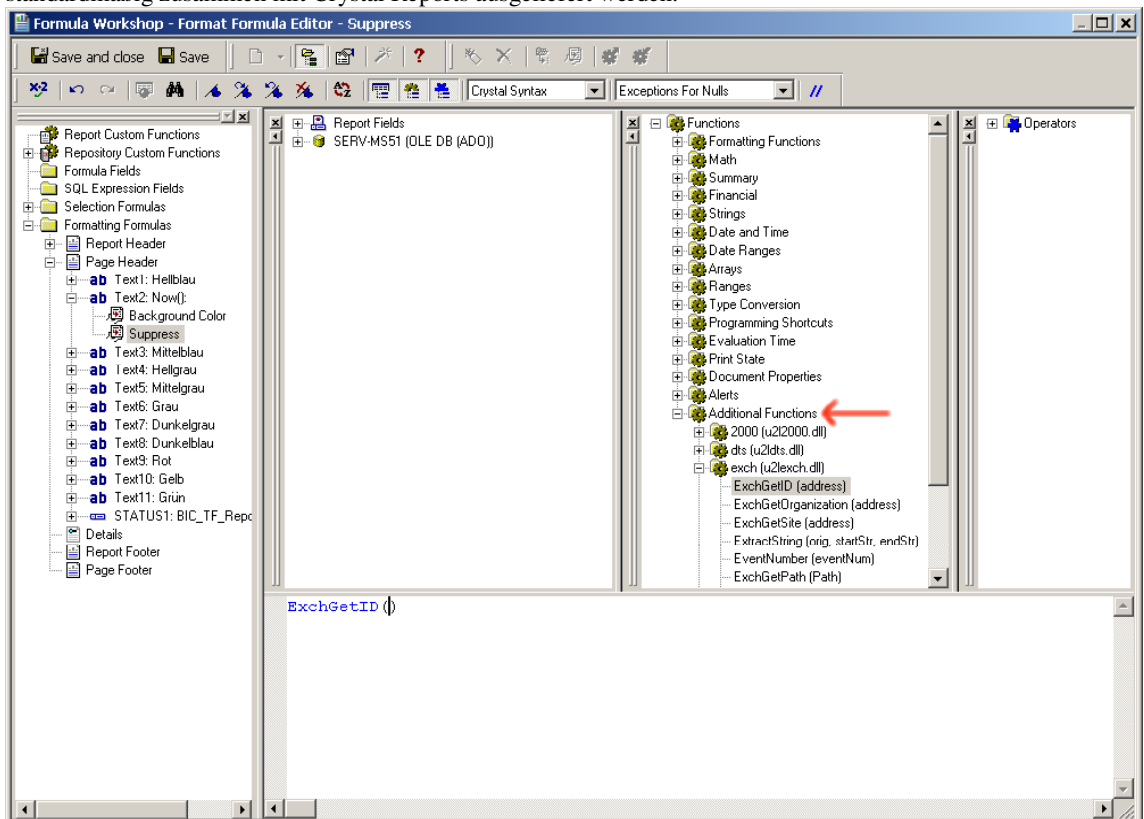
Juri Urbainczyk – 10.09.2007

Verwendung von User Function Libraries mit Crystal Reports	1
Was ist eine UFL?	1
Benutzung von UFLs	2
Erstellung von UFLs	3
Notwendige Dateien	3
Quelltext	4
Definition Datei	6
Erstellung der DLL	6
Implementierung von Farbkonstanten	6
Funktionen zur Laufzeit hinzufügen	8

Dieses Dokument beschreibt die Nutzung von User Function Libraries (UFLs) in Crystal Reports. Es werden ausschließlich mit C++ implementierte DLLs (dynamic link libraries) berücksichtigt.

Was ist eine UFL?

Viele Werte und Eigenschaften eines Berichts können in Crystal Reports dynamisch (also zur Laufzeit) berechnet werden. Die Hintergrundfarbe eines Objektes kann z.B. abhängig vom dargestellten Wert geändert werden. Die für diese Berechnung notwendige Formel kann im sogenannten „Formula Workshop“ („Formeleditor“) erstellt und gepflegt werden. Die Formel kann unter Verwendung von vorgefertigten Funktionen in Crystal oder Basic Syntax erstellt werden. Die existierenden Funktionen stellt der Formeleditor in einem Baum dar. Ein Doppelklick auf einen Funktionsnamen im Baum übernimmt ein entsprechendes Funktionstemplate (z.B. mit Platzhaltern für die Parameter) in den Editor. Dabei gibt es einen Knoten „Additional Functions“ (s. Abbildung), in dem zusätzliche Funktionen dargestellt werden, die nicht standardmäßig zusammen mit Crystal Reports ausgeliefert werden.



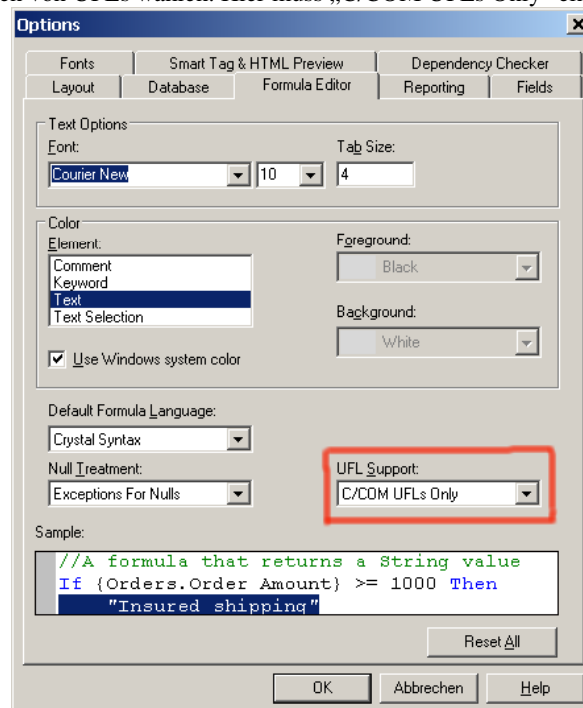
Diese zusätzlichen Funktionen können in Windows DLLs (dynamic Link libraries) ausgeliefert und Crystal zur Verfügung gestellt werden. Diese DLLs nennt man User Function Libraries (UFLs). DLLs enthalten ausführbaren Code, der zur Laufzeit in ein Programm eingebunden und aufgerufen werden kann. Aus einer UFL kann man prinzipiell auf alle Ressourcen des lokalen Betriebssystems zugreifen, für die der aktuelle Benutzer berechtigt ist. Daher handelt es sich um ein sehr mächtiges Werkzeug. UFLs können von der Business Objects Website (<http://support.businessobjects.com>) heruntergeladen, von Drittanbietern geliefert (z.B. <http://www.viksoe.dk/code/u21win32.htm>) oder auch selbst implementiert werden.

Der große Vorteil der UFL liegt vor allem darin, dass die dort definierte Funktion eben nur *an einer Stelle* implementiert ist und *in allen Reports* verwendet werden kann. Eine Änderung der Funktion wird sich (nach Installation der geänderten UFL) sofort auf alle Reports aus. Somit kann die UFL sogar als Ersatz für das Crystal Reports Repository fungieren.

Benutzung von UFLs

Um UFLs in seinem Report nutzen zu können, muss das System entsprechend konfiguriert sein. Folgende Punkte sind dabei zu berücksichtigen:

- Die zu verwendende UFL muss im korrekten Pfad abgelegt sein. Dieser ist
 - Client: „<drive>\Programme\Gemeinsame Dateien\Business Objects\3.0\bin“ oder „C:\Programme\Business Objects\Common\3.5\bin“ (je nach Installation)
 - Server: „<drive>\Common Files\Business Objects\3.0\bin“Die Pfade sind entsprechend anzupassen, je nachdem, ob es sich um eine Deutsche oder Englische Installation handelt, welche Crystal Reports Version verwendet wird, bzw. wo diese installiert ist.
- Der Name der DLL, die als UFL verwendet werden soll, muss mit U21 beginnen.
- Die UFL muss die für Crystal Reports notwendigen Funktionen exportieren (s. Erstellung von UFLs). Hat man eine bereits vorgefertigte UFL, ist dieser Punkt automatisch erfüllt. Beim Start von Crystal Reports bzw. beim Start des Webviewers werden alle DLLs die in den genannten Verzeichnissen liegen und die den Konventionen entsprechen, geladen und im Formeleditor zur Verfügung gestellt. Es ist nicht nötig, die DLL in der Registry zu registrieren.
- In den Optionen von Crystal Reports kann man im „Formula Editor“ Tab unter „UFL Support“ zwischen verschiedenen Arten von UFLs wählen. Hier muss „C/COM UFLs Only“ eingestellt sein (s. Abbildung).

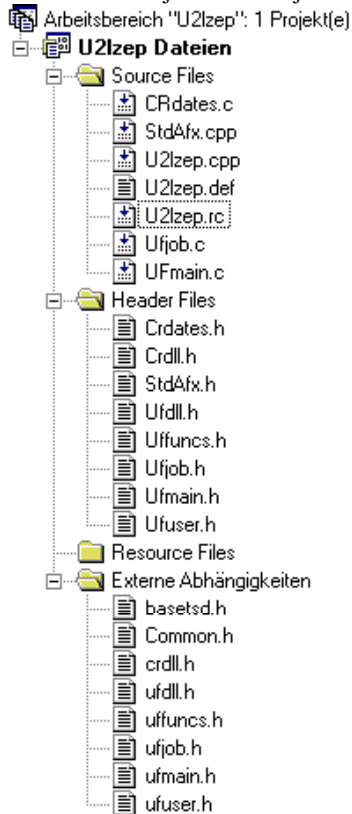


Sobald diese Voraussetzungen erfüllt sind, können UFLs mit Crystal Reports wie jede andere Funktion verwendet werden.

Erstellung von UFLs

DLL-UFLs können prinzipiell mit jeder COM-fähigen Programmiersprache (z.B. Visual Basic, Delphi, C, C++) erstellt werden. Im Folgenden wird die Erstellung mit Microsoft Visual C++ 6.0 beschrieben.

In Visual C++ legt man ein neues Projekt des Typs „Win32 Dynamic Link Library“ an oder kopiert ein bereits bestehendes Projekt. Das Projekt hat dann die folgende Struktur:



Notwendige Dateien

Die folgenden Dateien (s. Abbildung) werden nicht bearbeitet, müssen jedoch in das Projekt eingebunden werden. Sie gehören zur Developer Edition von Crystal Reports oder können auf der Website von Business Objects heruntergeladen werden.

File Name	Purpose
crdll.h	Defines primitives required by ufdll.h.
crdates.h	Declarations for functions to convert to and from Crystal Reports' date values.
crdates.c	Implementation of functions to convert to and from Crystal Reports' date values.
ufdll.h	Defines UFL enumerated, union, structure, and other data types.
ufuser.h	Prototypes of tables and functions user must implement
ufmain.h	Prototypes of internal UFL functions implemented in UFMAIN.C.
uffuncs.h	Prototypes of functions that must be exported by the UFL to be used by Crystal Reports. These functions are implemented in UFMAIN.C.
ufmain.c	Implementation of UFL functions used internally and used by Crystal Reports when connecting to the UFL. This file also contains generic LibMain and WEP functions for Windows 3.x DLLs and a generic DllEntryPoint function for Win32 DLLs. (The LibMain, WEP, and DllEntryPoint functions are defined conditionally according to whether or not you are building a Win32 DLL. You do not need to make any changes to the code here).
ufjob.h	Optional file. Contains a definition of the JobInfo structure (see UFJOB modules , and prototypes of the InifForJob function , TermForJob function), and FindJobInfo (see Implement InifJob and TermJob , implemented in UFJOB.C. structure and PROTOTYPES.C.)
ufjob.c	Optional file. Contains implementations of the required InifForJob function and TermForJob function . Also implements the FindJobInfo helper function.

Es müssen keine weiteren Bibliotheken oder Binärfiles dazugelinkt werden!

Quelltext

Der Quelltext der relevanten Funktionen wird in C++ geschrieben. Bei der Implementierung hat man vollen Zugriff auf den gesamten Sprachumfang von C++ sowie auf alle Funktionen der Win32-API. Die Funktionen können alle in einer cpp-Datei enthalten sein. Der erste Teil dieser Datei hat dann die folgende Struktur:

```
#include "stdafx.h"

//function declarations
ExternC UError CR_EXPORT TestFunction (UFParamBlock* ParamBlock);
ExternC UError CR_EXPORT ShortenString (UFParamBlock* ParamBlock);

//function table
UFunctionDefStrings FunctionDefStrings [] =
{
    {"String TestFunction ()", TestFunction},
    {"String ShortenString (String, Number)", ShortenString, ShortenStringRetLen},
    {NULL, NULL, NULL}
};

//editor templates
UFunctionTemplates FunctionTemplates [] =
{
    {"TestFunction"},
    {"ShortenString (!, )"},
    {NULL}
};

//function name to be shown in the tree
UFunctionExamples FunctionExamples [] =
{
    {"\tTestFunction"},
    {"\tShortenString (String,maxlen)"},
    {NULL}
};

ExternC UError CR_EXPORT ShortenStringRetLen(UFMemCalcBlock* MemCalcBlock)
{
    MemCalcBlock->UFMemCalcStringLength = 1024;
    return UNoError;
}
```

Das *Include*-Statement ist Standard und sollte nicht verändert werden. Es können weitere *includes* von eigenen Dateien hinzukommen, wenn man sich entscheidet, die Funktionalität auf verschiedene C++-Module (Dateien) aufzuteilen.

Als nächstes folgenden die Deklarationen der exportierten Funktionen. Wie im Beispiel sind sie immer mit der Signatur `ExternC UError CR_EXPORT` zu deklarieren und haben immer nur den einen Parameter `UFParamBlock* ParamBlock`.

Die folgenden Tabellen enthalten verschiedene für Crystal Reports notwendige Informationen über die in der UFL vorhandenen Funktionen. Die an dieser Stelle vorgestellte Implementierung ist statisch, d.h. dass Anzahl und Signatur der Funktionen zum Zeitpunkt der Kompilierung feststehen müssen und danach nicht mehr geändert werden können. Im Abschnitt Funktionen zur Laufzeit hinzufügen wird beschrieben, wie dynamische Funktionen implementiert werden können.

Im nächsten Block („function table“) befindet sich die Funktionstabelle, die Crystal Reports benötigt, um die Funktion zur Laufzeit aufrufen zu können. Die erste Spalte enthält die Funktionsdeklaration für Crystal Reports, also Returntype, Funktionsname und Parametertypen. Als Returntype und Parametertypen sind die Typen `String`, `Number`, `Boolean`, `DateTime` und die entsprechenden Arrays zulässig. Die zweite Spalte enthält den konkreten Pointer auf die Funktion, also die Adresse, die zur Laufzeit angesprungen wird. Hier ist darauf zu achten, dass tatsächlich Deklaration und Sprungadresse übereinstimmen, da sonst verwirrende Resultate entstehen können. Eine optionale dritte Spalte enthält die Adresse der Funktion, welche die maximale Länge des Returnwerts berechnet (im Beispiel ist das „ShortenStringRetLen“).

Im nächsten Block „editor templates“ findet sich eine weitere Tabelle, wobei jede Zeile mit der entsprechenden Zeile in der Funktionstabelle korrespondiert. Hier wird für jede Funktion ein Template hinterlegt, das in den

Crystal Formeeditor übernommen wird, wenn die Funktion z.B. durch Doppelklick ausgewählt wird. Typischerweise entspricht dies dem Funktionsnamen mit den Parametern. Hier könnte man auch längere Texte hinterlegen, die dann über den Funktionsnamen referenziert werden.

Optional kann man hier einen sogenannten „Insertion Point“ hinterlegen, das ist die Stelle, an der im Editor nach Einfügen des Textes der Cursor steht. Dazu fügt man an der gewünschten Stelle ein Ausrufezeichen ein.

Im nächsten Block „function name to be shown in the tree“ findet sich eine weitere Tabelle, wobei jede Zeile mit der entsprechenden Zeile in der Funktionstabelle korrespondiert. Für jede Funktion wird hier ein Text hinterlegt, der im Funktionsbaum des Formeeditors angezeigt wird. Typischerweise ist das der Name der Funktion, wobei ein Tabulator („\t“) vorangestellt wird.

Anschließend muss man eine Tabelle für mögliche Fehlermeldungen hinterlegen, was z.B. so aussehen kann:

```
char* ErrorTable [] =
{
    /* 00 */ "no error",
    /* 01 */ "Wrong parameter type",
    /* 02 */ "Invalid date",
};
```

Die Fehlermeldungen können dann in den eigenen Funktionen verwendet werden (s. unten).

Im folgenden Abschnitt der Quelldatei findet man zwei Standardfunktionen, die jede UFL besitzen muss:

```
// Management functions
void InitJob(struct JobInfo*)
{
}

void TermJob(struct JobInfo*)
{
}
```

Diese beiden Funktionen werden laut Dokumentation von Business Objects ausgeführt, „bevor ein Job gedruckt wird bzw. nachdem er gedruckt wurde“. Was auch immer das bedeutet, man kann auf jeden Fall in die Funktion „InitJob“ solche Aktionen aufnehmen, die mindestens einmal *vor* der Ausführung irgendeiner anderen Funktion durchgeführt werden sollten (z.B. das Initialisieren von DLL-lokalen Variablen).

Im Weiteren kann man nun seine eigenen Funktionen implementieren. Im folgenden Beispiel implementieren wir zwei Funktionen, wobei die eine nur eine Zeichenkette zurückgibt und die andere Funktionen einen Wert aus einer Datei liest.

```
// Function library

ExternC UFEError CR_EXPORT TestFunction(UFParamBlock* ParamBlock)
{
    strcpy( ParamBlock->ReturnValue.ReturnString, "Dies ist ein String" );
    return UFNoError;
}

//cut a string after maxlength chars and add "...":
ExternC UFEError CR_EXPORT ShortenString(UFParamBlock* ParamBlock)
{
    //get first parameter, which is the string
    UFParamListElement* FirstParam = GetParam(ParamBlock, 1);
    if( FirstParam == NULL ) return UFNotEnoughParameters;
    if( FirstParam->ParamType != UFStringArg ) {
        ParamBlock->ReturnValue.UFReturnUserError = 1;
        return UFUserError;
    }
    char* MyString = FirstParam->Parameter.ParamString;

    //get second parameter, which is the max length
    UFParamListElement* SecondParam = GetParam(ParamBlock, 2);
    if( SecondParam == NULL ) return UFNotEnoughParameters;
    if( SecondParam->ParamType != UFInteger && SecondParam->ParamType != UFNumber )
    {
        ParamBlock->ReturnValue.UFReturnUserError = 1;
        return UFUserError;
    }
}
```

```

//correct for strange CR number format
int maxlength = SecondParam->Parameter.ParamNumber/100;

int length = strlen(MyString);

char buffer[1024];
memset( buffer, 0, 1024 );

if (length<1024) {
    strcpy( buffer, MyString);
}

if ((length>maxlength) && (length<1021)) {
    strcpy( buffer+maxlength, "..." );
}

strcpy(ParamBlock->ReturnValue.ReturnString, buffer);

return UFNoError;
}

```

Wie im Quelltext ersichtlich greift man auf die Parameter einer Funktion immer über `GetParam(ParamBlock, <nr>)` zu. Return-Werte vom Typ String werden mit `strcpy(ParamBlock->ReturnValue.ReturnString, "Text...")` gesetzt. Bei anderen Typen ist z.B. auch `ParamBlock->ReturnValue.ReturnBoolean` oder `ParamBlock->ReturnValue.ReturnNumber` möglich.

Die Funktion `ShortenString()` im Beispiel, liest zwei Parameter: einen String und eine Zahl „maxlength“. Wenn der String mehr als „maxlength“ Zeichen besitzt, wird er abgeschnitten und um den String „...“ ergänzt.

Definition Datei

Jedes DLL-Projekt muss eine Datei mit der Endung `def` enthalten. Diese Datei muss folgendes Format besitzen:

```

LIBRARY <name der DLL ohne .dll-Endung>
DESCRIPTION ' <ein beschreibender Text>'
EXPORTS
    UFInitialize
    UFTerminate
    UFGetVersion
    UFStartJob
    UFEndJob
    UFGetFunctionDefStrings
    UFGetFunctionExamples
    UFGetFunctionTemplates
    ULErrorRecovery
    TestFunction
    ShortenString

```

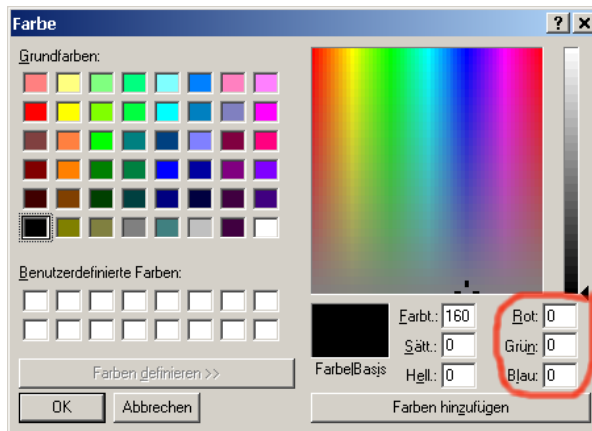
Wichtig ist vor allem, dass die mit `UF` beginnenden Funktionen in der Definition Datei enthalten sind. Es handelt sich dabei um Standardfunktionen, die jede UFL besitzen muss, da sonst Crystal Reports die DLL nicht als UFL erkennt.

Erstellung der DLL

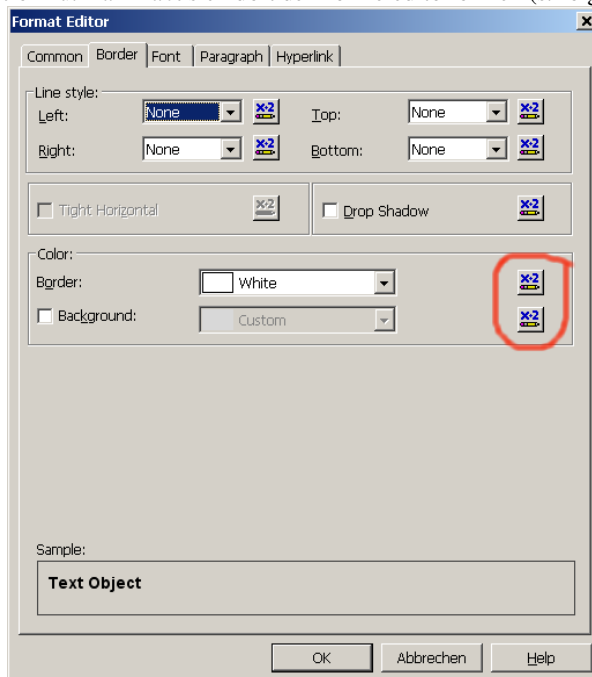
Mit dem Menüpunkt „Erstellen -> Alles neu erstellen“ wird ein komplettes Kompilieren aller Source Code Dateien angestoßen. Wenn alles ohne Fehler durchläuft, wird am Ende die entsprechende DLL-Datei im Unterverzeichnis „Debug“ bzw. „Release“ erzeugt.

Implementierung von Farbkonstanten

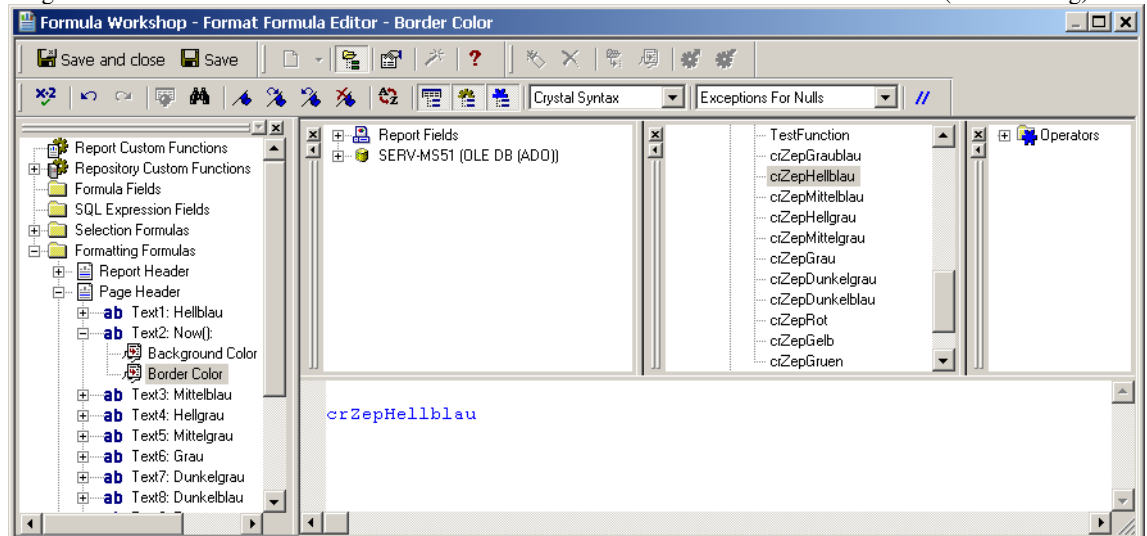
Ein Problem bei Crystal Reports ist, dass Farben, die man häufig verwendet, nicht über eine Crystal Report Sitzung hinaus in einer Farbtabelle abgelegt werden können. Man muss dann die Farben immer wieder neu im Dialog für die Farbauswahl zusammenstellen. Dazu muss der Entwickler die RGB-Werte der Farben kennen (s. Abbildung).



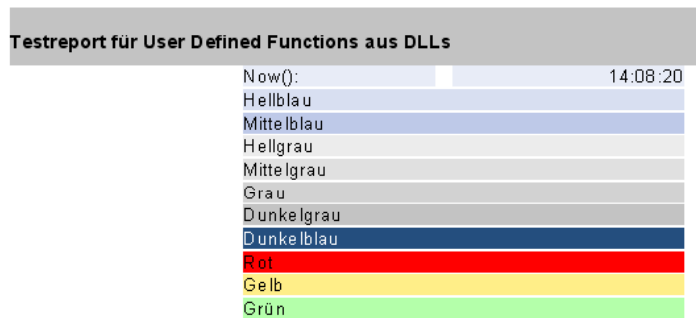
Allerdings läßt Crystal Reports an nahezu allen Stellen, wo Farben angegeben werden können, auch die Farbbestimmung per Funktion zu. Dann läßt sich dort der Formeleditor öffnen (s. folgende Abbildung).



Die Farben lassen sich also auch über entsprechende Funktionen in UFLs definieren, die als Farbkonstanten fungieren. Diese Funktionen lassen sich dann über den Formeleditor als Farbwert auswählen (s. Abbildung):



In einem Report kann man die Farbfunktionen aus der UFL z.B. verwenden, um die Hintergrundfarben für verschiedene Textobjekte zu setzen (s. Abbildung).



Die Verwendung einer UFL an dieser Stelle bietet folgende Vorteile:

- Die „Farbkonstanten“ ersparen dem Entwickler das Nachschlagen und Eintippen der Farbwerte.
- Eine nachträgliche Änderung der Farbwerte wirkt sich sofort auf alle Reports, ohne dass diese einzeln neu angefaßt werden müssen.

Funktionen zur Laufzeit hinzufügen

Sobald man Farbkonstanten als Funktionen in der UFL implementiert, möchte man die Farbwerte leicht nachträglich ändern können. Zudem möchte man weitere Farbkonstanten einfach hinzufügen können, ohne die DLL jedesmal neu übersetzen zu müssen. Auch das ist möglich.

Sowohl die Farbwerte (RGB) als auch die Namen und Templates der Funktionen werden aus einer INI-Datei gelesen, die `functions.ini` heißt. Deren Inhalt sieht z.B. so aus:

```
// color <funcName> <red> <green> <blue> +<template text to be pasted in editor>
color crColGraublau      232 236 247      +crColGraublau // 232 236 247
color crColHellblau     216 223 241      +crColHellblau // 216 223 241
color crColMittelblau   190 201 232      +crColMittelblau // 190 201 232
color crColHellgrau     236 236 236      +crColHellgrau // 236 236 236
color crColMittelgrau   224 224 224      +crColMittelgrau // 224 224 224
color crColGrau         210 210 210      +crColGrau // 210 210 210
color crColDunkelgrau   195 195 195      +crColDunkelgrau // 195 195 195
color crColDunkelblau   36 78 126      +crColDunkelblau // 36 78 126
color crColRot          255 0 0        +crColRot // 255 0 0
color crColGelb         255 238 136     +crColGelb // 255 238 136
color crColGruen        180 255 170     +crColGruen // 180 255 170
color crSchwarz         0 0 0          +crSchwarz // 0 0 0
```

In der INI-Datei stellt jede Zeile, die mit dem Schlüsselwort „color“ beginnt, eine Farbkonstante dar. Nach dem Schlüsselwort folgt der Name der Funktion, dann die drei Farbwerte (RGB) und dann, eingeleitet von einem Pluszeichen, das Template, also der Text, der in den Formeleditor eingefügt wird.

Eine UFL enthält verschiedene Standardfunktionen, die von Crystal Reports verwendet werden, um die Funktionen der UFL zu ermitteln. Die Funktion `UFGetFunctionExamples()` wird von diesen als erstes aufgerufen. In dieser Funktion wird nun die Funktion `getFunctionsFromFile()` aufgerufen:

```
ExternC UFFunctionExampleList FAR * CR_EXPORT UFGetFunctionExamples ()
{
    int iFunctionCount
        = getFunctionsFromFile( <INI-filename and path>,
            &FunctionExamples, &FunctionTemplates, &FunctionDefStrings );

    static UFFunctionExampleList FunctionExampleList =
    {
        {sizeof (UFFunctionExampleList)},
        {FunctionExamples}
    };

    return (&FunctionExampleList);
}
```

In der Funktion `getFunctionsFromFile()` wird das Lesen der INI-Datei implementiert:

```
int getFunctionsFromFile( char* filename, UFFunctionExamples** fe,
```

```

UFFunctionTemplates** ft, UFFunctionDefStrings **fd ) {

*ft = (UFFunctionTemplates*) malloc( sizeof(UFFunctionTemplates) );
(*ft)[0].FuncTemplate = NULL;

*fe = (UFFunctionExamples*) malloc( sizeof(UFFunctionExamples) );
(*fe)[0].FuncExample = NULL;

*fd = (UFFunctionDefStrings*) malloc( sizeof(UFFunctionDefStrings) );
(*fd)[0].FuncDefString = NULL;
(*fd)[0].UFEntry = NULL;
(*fd)[0].UFMemCalcFunc = NULL;

char buffer[256];

FILE *f = fopen( filename, "r");

if (f != NULL) {
    while(!feof(f))
    {
        memset(buffer, 0, 256 );
        fgets(buffer, 255, f);

        //only start when line is not empty and starts with 'color'
        if ((strlen(buffer)>1) && (strstr(buffer,"color") == buffer) ) {

            char functionName[64];

            int r = 0, g = 0, b = 0;

            functionCount ++;

            //skip over the 'color '
            strcpy( buffer, buffer+strlen("color")+1 );
            sscanf( buffer, "%s %d %d %d", &functionName, &r, &g, &b );

            //the template text should start after a plus sign:
            char* functionTemplate = strchr(buffer,'+');
            if (functionTemplate == NULL) {
                functionTemplate = functionName;
            } else {
                functionTemplate ++;
            }

            char* funcExample = getExampleString( functionName, r, g, b );

            colors[functionCount-1].r = r;
            colors[functionCount-1].g = g;
            colors[functionCount-1].b = b;

            addFunction( functionName, functionTemplate, funcExample,
                "Number", "()", myColor, NULL, fe, ft, fd );
        }
    } // ENDWHILE

    fclose(f);
}

return functionCount;
}

```

In der Funktion `addFunction()` wiederum, die für jede relevante Zeile der INI-Datei aufgerufen wird, werden die für die UFL benötigten Tabellen mit den Funktionsinformationen jeweils dynamisch erweitert und korrekt besetzt:

```

void addFunction( char* functionName, char* functionTemplate, char*
functionExample, char* functionType, char* parameter,
UFError ( __stdcall *funcPtr)(UFParamBlock* ParamBlock),
UFError ( __stdcall *retLenFuncPtr)(UFMemCalcBlock* MemCalcBlock),
UFFunctionExamples** fe, UFFunctionTemplates** ft,

```

```

        UFFunctionDefStrings **fd )
{
    UFFunctionTemplates* ft1 = (UFFunctionTemplates*) malloc(
        sizeof(UFFunctionTemplates)*(functionCount+1) );
    memmove(ft1, (*ft), sizeof(UFFunctionTemplates)*functionCount );
    ft1[functionCount-1].FuncTemplate = strdup(functionTemplate);
    ft1[functionCount].FuncTemplate = NULL;
    free(*ft);
    *ft = ft1;

    UFFunctionExamples* fe1 = (UFFunctionExamples*) malloc(
        sizeof(UFFunctionExamples)*(functionCount+1) );
    memmove(fe1, (*fe), sizeof(UFFunctionExamples)*functionCount );
    fe1[functionCount-1].FuncExample = functionExample;
    fe1[functionCount].FuncExample = NULL;
    free(*fe);
    *fe = fe1;

    UFFunctionDefStrings* fd1 = (UFFunctionDefStrings*) malloc(
        sizeof(UFFunctionDefStrings)*(functionCount+1) );
    memmove(fd1, (*fd), sizeof(UFFunctionDefStrings)*functionCount );
    fd1[functionCount-1].FuncDefString = getFunctionDefString( functionName,
        functionType, parameter );
    fd1[functionCount-1].UFEntry = funcPtr;
    fd1[functionCount-1].UFMemCalcFunc = retLenFuncPtr;

    fd1[functionCount].FuncDefString = NULL;
    fd1[functionCount].UFEntry = NULL;
    fd1[functionCount].UFMemCalcFunc = NULL;
    free(*fd);
    *fd = fd1;
}

```

Der wesentliche Parameter ist dabei „funcPtr“, also der Callback der konkreten Funktion, die für die neue Funktion zur Ausführungszeit tatsächlich aufgerufen werden soll. In diesem Beispiel ist es die Funktion `myColor()`, in der aus dem Array „colors“, die für diese Funktion aus der INI-Datei gespeicherten RGB-Werte zurückgegeben werden:

```

ExternC ULError CR_EXPORT myColor(UFParamBlock* ParamBlock)
{
    int i = (ParamBlock->FunctionNumber)-functionOffset;

    ParamBlock->ReturnValue.ReturnNumber = RGB(colors[i].r, colors[i].g,
        colors[i].b) * UFNumberScalingFactor;

    return UFNoError;
}

```

Die Konstante `functionOffset` in der Funktion `myColor()` ist die Position der ersten Funktion der UFL in der internen Funktionstabelle von Crystal Reports. Zum Zeitpunkt der Ausführung von `myColor()` wird dieser Wert von der Nummer der aktuell ausgeführten Funktion abgezogen und somit das Offset der zu bildenden Farbe im Array `colors` ermittelt.

Diese Konstante ist fest, solange keine weitere UFL zur Installation hinzugefügt wird, deren Funktionen *vor* der hier beschriebenen UFL gelesen werden. Eine Änderung ist auch mit einer neuen Version von Crystal Reports möglich, wenn auch unwahrscheinlich. Um eine einfache Änderung dieser Konstante zu erlauben wird auch sie aus dem INI-File gelesen.