Utvärdering av Windows RT för portning av Mario Framework

Marcus Isaksson

KTH Industriell teknik
och management

Innehållsförteckning

Bakgrund

Examensjobbet

Ericsson har utvecklat ett ramverk för multimediakommunikation som heter Mario. Mario kan användas för att sätta upp video- och ljud-samtal över IP och finns fungerande på Android och iOS. Mario är en del av web browsern Bowser som finns att hämta på Google Play[i] eller App Store[ii]. Bowser möjliggör att köra WebRTC applikationer skrivna i HTML5 och JavaScript på Android och iOS.

Nu när Microsoft släpper sina nya plattformar, Windows RT, Windows 8 och Windows Phone 8 så finns ett intresse att försöka flytta detta ramverk till dessa plattformar. Vi kommer göra ett försök att porta Mario till Windows RT. Anledningen till att Windows RT valdes är för att det är den första plattformen att släppas av de tre nämnda produkterna.

Plattforms APIet som används i Windows RT kallar för Windows Runtime (WinRT).[iii] Många delar av det här APIet delas med Windows Phone 8 och hela WinRT APIet exponeras i Windows 8. Det här betyder att förutsatt att koden designas på rätt sätt så borde den gå att flytta mellan Windows RT, Windows 8 och Windows Phone 8 utan problem. Windows RT är ett operativsystem utvecklat för att fungera på datorer som använder ARM-baserade processorer.

För att Mario ska kunna fungera på WinRT så måste följande mål uppfyllas:

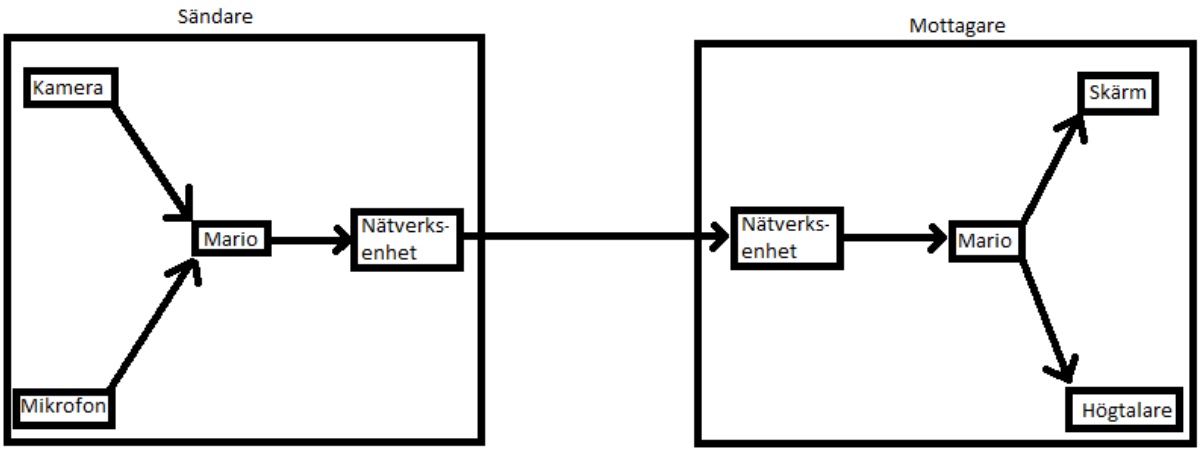
- De open-source bibliotek som används av Mario måste byggas på Mario.
- C/C++ kod måste kunna köras på Windows RT plattformen.
- Mario ramverket måste integreras med video- och ljudfångning i operativsystemet.

Vad det gäller open-source projekt så ligger det under en sådan licens (LGPL) som gör att källkoden inte får ändras utan att hela källkoden till de projekten som använder open-source koden också släpps som open-source. Ericsson vill inte att Mario ska släppas som open-source utan vill själva ha kontroll på källkoden. Av denna anledning kommer vi utreda om det är möjligt att bygga dessa bibliotek utan att ändra i koden.

I och med att Mario är skrivet i C och C++ så måste det också utredas hur vida det är möjligt att köra C/C++ kod i Windows RT. Vad det gäller video- och ljudfångning måste Windows RT APIet utredas för att se hur man använder det för just detta.

Mario

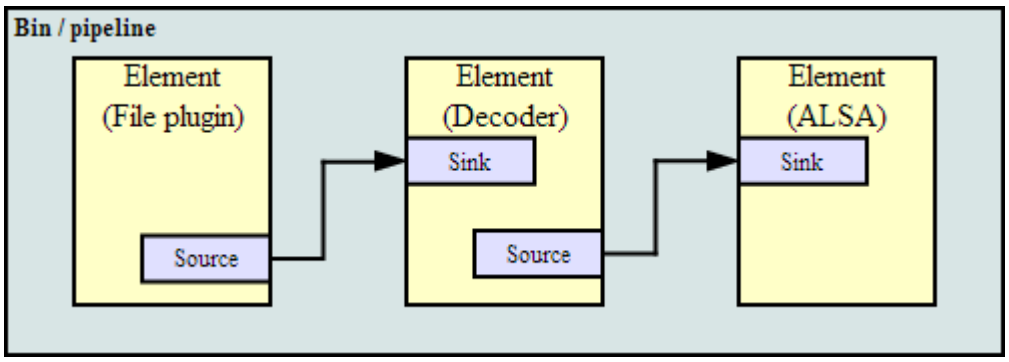
Mario är som sagt ett multimedia-ramverk. Det tar hand om ljud- och videoupptagning, uppkopplingar i samtal och skickar data mellan alla uppkopplade. Mario kan ta in video och ljudströmmar från mikrofoner och kameror på de plattformar den är implementerad på och koda om det här data med ett antal olika kodexar. Mario kan även avkoda denna data på mottagarsidan till en bitmap som kan visas på skärmen. De kodexar som stöds i dagsläget är VP8 och H.264 för video och G.711 för ljud. Mario är fortfarande under utveckling så stöd för fler kodexar tillkommer löpande. Här under följer en beskrivning över var Mario gör i ett simpel sändare/mottagare scenario.



Som ni ser finns Mario hos både sändaren och mottagaren. Det Mario gör är att hämta en videoström och en ljudström från mikrofonen och kameran på den devicen som den körs på. Koda om de här strömmarna till nått som är lämpligt att skicka över ett nätverk och skicka det via en nätverksenhet till mottagaren. I mottagaränden skickar nätverksenheten dessa strömmar till Mario. Mario kodar av dem och videon kan renderas på skärmen och ljudet spelas upp i högtalaren. Mario ser även till att själva nätverksströmmen sätts upp och upprätthålls under hela samtalet.

GStreamer

Mario bygger på open-source biblioteket GStreamer. GStreamer är ett ramverk som underlättar behandling av media och är byggd på en pluginarkitektur och bygger på att man sätter olika plugins i en kedja för att avkoda eller koda ljud och videostömmar. Här under följer ett diagram på en simpel MP3 spelare framtagen med hjälp av GStreamer.



Här ser vi att vi använder oss av 3 olika element för att framställa MP3 spelaren. De två första elementen är plugins som finns i GStreamer ramverket. Den fösta läser MP3 filen från hårddisken och skickar in den i decoder-plugginen. Den här kodar av filen till ett format som kan skickas in i ALSA ljuddrivrutinen (det sista elementet) som spelar upp ljudet i högtalaren. Den här följden av plugins kallas för en media graf.

Elementen i en graf kommunicerar med varandra igenom nått som kallas för pads. Det finns två olika typer av pads, source och sink. En source-pad kan liknas med en output och där igenom skickas data ut ur pluginen. På en sink-pad kan man koppla på en source-pad och på så sätt kedja ihop flera olika plugins.

Den här pluginbaserade arkitekturen gör det otroligt lätt att lägga till stöd för nya kodexar och filtyper. Detta har gjort att GStreamer snabbt plockats upp av många företag som producerar media applikationer. GStreamer används i allt från media spelare till realtidskommunikationsapplikationer

Förförande

Utvärdering av Windows RT

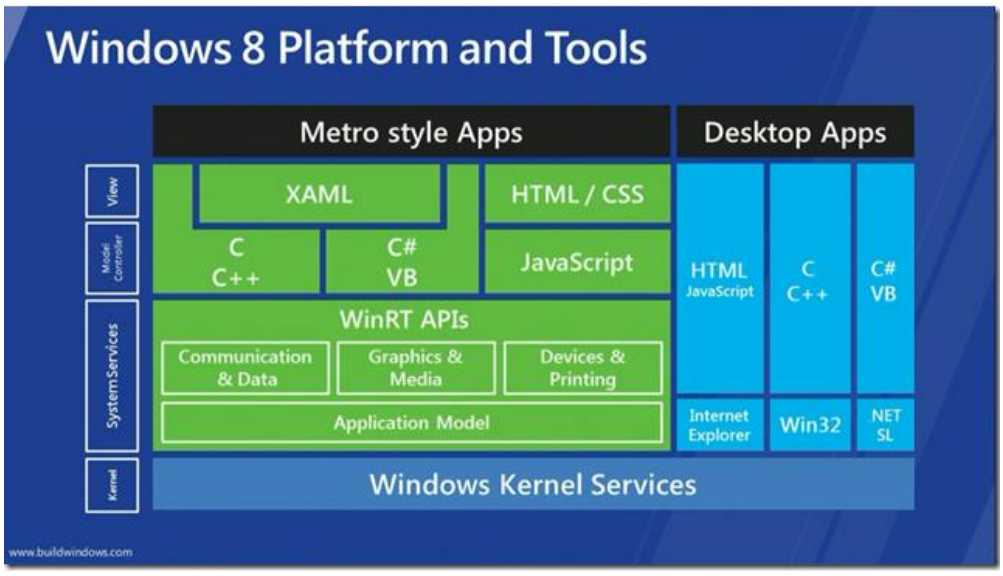
I och med att Windows RT dels är ett nytt operativsystem och också använder ett helt nytt API för operativsystemsfunktioner så måste vi först och främst gå igenom hur Windows RT skiljer sig från andra plattformar när man programmerar. Windows RT är i första hand tänkt som ett operativsystem för tablets och där av är de språken som kan användas också anpassade för utveckling av appar och det ställer andra krav än utvecklingar.

Windows RT är det nya operativsystemet från Microsoft som kör på en ARM processor arkitekturen. Till Windows RT utvecklades det nya Windows Runtime APIet som ersätter det klassiska Win32 APIet för system anrop. WinRT innehåller ett subset av den funtionallitet och mycket går att använda precis som i Win32. Det finns dock vissa funktioner som är ändrade och som måste behandlas med nya metoder. Bland annat trådhantering, filhantering, sockets och grafik har designats om i det nya APIet. Den här funktionaliteten exponeras via C++ Component Extensions (C++/CX) som är ett tillägg Microsoft gjort till C++ för att kunna anropa C++ kod ifrån olika .NET språk.

Alla C++ bibliotek du skriver kompileras till speciella DLLer som kan köra på Windows RT, de här biblioteken kan bara anropas av applikationer skrivna i C++/CX med DirectX eller av en Windows Runtime Component skrivna i C++/CX. En Windows Runtime Component är en komponent som kan användas av Windows Store Applikationer oavsett vilket språk de är skrivna i. En Windows Runtime Component kan skrivas i vilket .NET språk som helst men den kan bara anropa native C++ DLLer om den är skriven i C++/CX. När koden kompileras skapas metadata automatiskt som gör det möjligt för alla .NET språk att anropa den här komponenten.[iv]

Kod som fungerar på Win32 och inte använder några utav den Win32 funktionalitet som ändrats samt går att kompilera med Microsoft Visual C++ går att köra på Windows RT. Vad det gäller applikationsutveckling så kan applikationer skrivas i C++/CX med DirectX, C#, VisualBasic och JavaScript/HTML5.

Med hjälp av Windows Runtime Components kan man alltså wrappa sina bibliotek, förutsatt att biblioteket går att bygga på Windows RT och på så sätt nå alla sina C++ bibliotek från vilket .NET språk du än väljer att använda till din applikation.



Beskrivning av WinRT kontra Win32.

När WinRT designades så bestämde man för att se till att applikationer blir responsiva och inte ”hänga sig” göra många av de API anropen måste anropas asynkront. I C# kan man använda nyckelordet `async` för att skapa en metod som kan köras i bakgrunden medan resten av applikationen fortsätter köra. Det finns vissa asynkrona funktioner som man vill vara säker på att de slutförs innan man går vidare i koden och då kan man använda nyckelordet `await`. Om du sätter det ordet innan ett anrop till en metod eller metod som asynkron så kommer man vänta på att funktionsanropet är klart och returnerar innan man går vidare i applikationen. För att man ska få använda `await` i en funktion måste funktionen vara asynkron.[v]

I C++/CX kan skapa en `task` som kan köras i bakgrunden. Alla anrop som ska köras asynkront returnerar en `IAsyncOperation` som är en operation som kan köras asynkront i bakgrunden. I C++ namespacet `concurrensy` finns en hjälpfunktion för att skapa tasks som heter `create_task` som tar en `IAsyncOperation` som parameter.[vi]

Skapa en byggmiljö för de open source projekt Mario bygger på

Mario bygger på ett antal olika open source projekt. Dessa måste byggas för Windows RT för att Mario över huvud taget ska kunna köras på Windows RT. Eftersom att alla open source projekt som Mario bygger på är licensierade under LGPL (Lesser General Public License) så får koden i dessa projekt inte ändras utan att hela Mario måste publiceras som open source och det ligger inte i Ericsson intresse. Vi måste alltså hitta ett sätt att kompilera och köra koden utan att ändra den. Den enda kompilatorn som just nu kan bygga kod för Windows RT är Microsoft Visual C++. Vad det gäller byggandet av dessa projekt så finns det två stora problem.

Problem 1, Windows Runtime

Själva Mario i sig borde vara lätt att bygga på Windows RT då den innehåller väldigt lite plattformsspecifik kod. Däremot finns det en hel del plattformsspecifik kod i de bibliotek Mario bygger på. GStreamer har en hel del kod som måste portas och skivas om för att det ska fungera på Windows RT. Det här betyder att för att det ska vara möjligt att bygga på Windows RT måste först GStreamer gå igenom stora omskrivningar där Win32 koden gås igenom och skrivs om för att fungera på Windows RT.

Det största problemet är att Windows RT inte bara är en ny version utav Windows utan att hela APIet har om designats från grunden. Det är helt enkelt för stora skillnader mellan Windows Runtime APIet och Win32 APIet. Det finns inte heller några sätt att återanvända Win32 kod i Windows RT.

Problemet slutar inte vid GStreamer, utan även det bygger på projektet GLib som också har plattformsspecifik kod. I GLib finns det bland annat en IO implementation som använder sig av winsock2 (Socket strukturen som användes i Win32) för socketkommunikation, den här måste skrivas om då förförandet att öppna och använda sockets har ändrats. Det finns även filhanteringsfunktioner och trådfunktioner som måste skrivas om. På grund av licenser på dessa bibliotek så får jag inte ändra källkoden i dessa bibliotek.

Problem 2, Bygga med Microsoft Visual C++

GStreamer har ingen bra support för att bygga med Microsoft Visual C++. Det fanns ett gammalt projekt som hette OSSBuild project som var ett sätt att bygga GStreamer med Microsoft Visual C++ men det är tyvärr övergivet. Just nu går GStreamer bara att bygga med GNU Compiler Collection (GCC). Det här ställer till stora problem då den enda kompilatorn som kan bygga applikationer för Windows RT är Microsoft Visual C++ kompilatorn och just nu finns det inget stöd att bygga koden med nått annat än GCC.

Slutsats

Eftersom att koden måste skrivas om innan vi kan bygga den och jag inte får göra det så kan jag inte skapa en byggmiljö för open source biblioteken och det underminerar hela projektet. Om en portning ska kunna göras inom de ramar som satts upp för projektet så måste vi först vänta in en uppdatering av GStreamer som fungerar på Windows RT. Vissa nyckelpersoner som utvecklar GLib har utryckt intresse för att porta GLib till WinRT men det finns inga uppgifter om när en port kommer finnas tillgänglig.[vii] I och med att vi inte kan göra någon byggmiljö och därför inte kan porta Mario till Windows RT kommer resten av rapporten gå igenom tekniker för att använda och kalla på C/C++ kod i Windows RT appar samt en genomgång på hur videoupptagning fungerar i Windows RT.

Wrappning av C++ kod i Windows RT

Mario är skrivet i C/C++ men applikationer som skrivs för Windows RT måste skrivas i språken C++/CX, C#, Visual Basic eller JavaScript. C++/CX är väldigt likt C++ men det skiljer så mycket mellan dessa språk så att man inte bara kan dra in C++ kod i ett C++/CX projekt och kompilera det. Hur man hanterar pekare är ändrat bland annat. Vi måste därför hitta ett sätt att kunna köra och anropa C/C++ kod från något av dessa språk. Annars kommer Mario inte kunna användas i applikationer som skrivs för Windows RT.

Att wrappa ett C++ bibliotek för användning i App

Först och främst måste vi utreda om det finns tekniker för att kompilera och anropa C/C++ kod från något av de språken som man kan skriva Windows RT appar i. Det går att skriva helt vanliga bibliotek i C++ och kompilera för Windows RT. Nackdelen med dessa bibliotek är att de bara kan använda av en applikation som är skriven i C++/CX. Det här kan ställa till det för en applikationsutvecklare då de inte kan använda det språket de känner sig mest hemma i utan blir tvingade till ett av de fyra språken som kan användas.

I Windows RT finns det något som kallas Windows Runtime Components. En Windows Runtime Component är ett bibliotek som kan skrivas i vilket utav Windows RT språken som helst och som kompileras till ett bibliotek som kan användas av alla Windows RT språk. Eftersom C++/CX kod kan anropa C++ kod så kan vi skriva en Windows Runtime Componenet i C++/CX, anropa C++ kod i den och sen anropa komponenten via en Windows RT applikation oavsett om den är skriven i C++/DirectX, C#, VB eller JavaScript. På så vis möjliggör vi användning av vanlig C/C++ kod i Windows RT applikationer.

```
#pragma once

//NativeDll.h
namespace NativeDll
{
    class Funcs
    {
    public:
        __declspec(dllexport) Funcs();
        __declspec(dllexport) ~Funcs();
        __declspec(dllexport) int Add(int i, int j);
    };

Funcs::Funcs() { }

Funcs::~Funcs() { }

}
```

```
// NativeDll.cpp : Defines the exported functions for the DLL
application.

#include "pch.h"
#include "NativeDll.h"

using namespace NativeDll;

int Funcs::Add(int i, int j)
{
    return i + j;
}
```

Det här ovan kompileras till en helt vanlig Dll som kan köras på WinRT. Det här är helt vanlig C++ kod. Det enda vi har i .cpp filen är en funktion som tar emot två integers, adderar dem och returnerar vad resultatet blev. Den här koden är helt plattformsoberoende. I header filen har vi definierat Add funktionen som public samt den tomma konstruktorn och destruktorn. Vi har även använt oss av `__declspec(dllexport)` flaggan. Den här flaggan säger till kompilatorn exportera dessa filer när DLLen skapas så att den kan anropas av program som använder DLLen. Det här är den enda delen av koden som är plattformsspecifik.

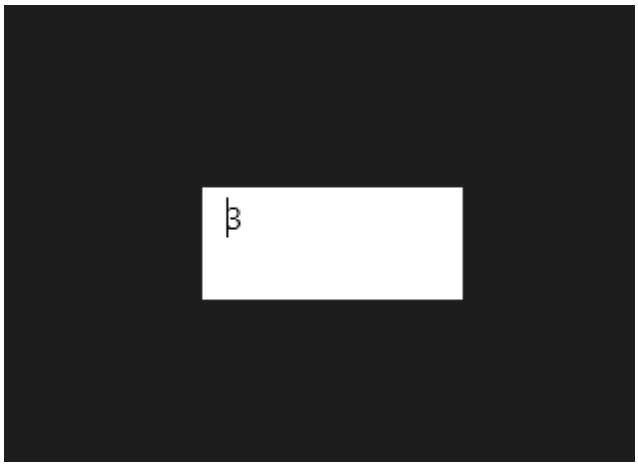
```
//CXWrapper.h

#pragma once

namespace CXWrapper
{
    public ref class MathFuncs sealed
    {
    public:
        MathFuncs();
        int Add(int i, int j);

    private:
        ~MathFuncs();
    };
}
```

```
// CXWRapper.cpp

#include "pch.h"
#include "CXWrapper.h"
#include "NativeDll.h"

using namespace CXWrapper;
using namespace NativeDll;

MathFuncs::MathFuncs(){}

MathFuncs::~MathFuncs(){}

int MathFuncs::Add(int i, int j)
{
    Funcs* funcs = new Funcs();
    int retVal = funcs->Add(i,j);
    delete(funcs);
    return retVal;
}
```

Koden här ovan kompileras till en Windows Runtime Component som kan användas av en Windows Store applikation. Klassen markeras med ref för att den ska få egenskaper som referensräkning och då kan användas i ett minneshanterat språk. Destuktorn måste vara privat för att klassen inte ska kunna destruktas från källkoden i minneshanterade språk. Klassen måste också markeras med sealed eftersom att den inte får ärvas. När klassen MathFuncs skapas så skapar den i sin tur en instans av klassen Funcs som finns i NativeDll. MathFuncs::Add anropar i sin tur Funcs::Add och på så sätt är den wrappad för använding i Windows Store applikationer.

```
using namespace CXWrapper;

namespace CXTest
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
            MathFuncs mathFuncs = new MathFuncs();
            MyTextBox.Text = mathFuncs.Add(1, 2).ToString();
        }
    }
}
```

Ovan är koden för en enkel Windows Store applikation skriven i C# som bara har en TextBox i sig kallad MyTextBox. Den har en referens till CXWrapper biblioteket och skapar en instans av MathFuncs och skriver ut returvärdet av MathFuncs.Add i TextBoxen. Här kan man se att även fast den anropar en C++ metod så använder man sig av C# syntaxen. Det är för att när Windows Store komponenten byggs så skapas en lista med hur alla olika .NET språk ska kunna använda den och den här listan skapas så ändrar den alla funktionsanrop så att den passar syntaxen för dessa språk.[viii]



Det här visar alltså att det är möjligt att anropa C/C++ kod ifrån en Windows RT app. Det här är otroligt viktigt om det ska gå att bygga en applikation som använder sig av Mario för Windows RT. Hela ramverket borde alltså gå att importera som ett Windows Runtime Library och kompileras. All funktionalitet som ska kunna användas direkt från applikationen måste så klart wrappas med C++/CX kod men som vi visat här är det ett ganska litet arbete. Tekniken med Windows Runtime Library kommer också underlätta vid en portning av GStreamer och GLib för att behålla så mycket av den C/C++ koden som möjligt.

Skillnader mellan native C/C++ kod och C++/CX

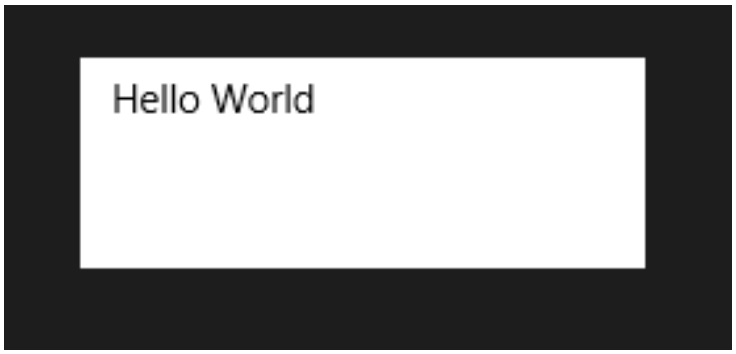
I och med a C++ är ett språk som inte är minneshanterat och de andra .NET språken är det så har man i C++/CX lagt till en ny typ av pekare som gör det lättare att skicka objekt in och ut ur C++/CX kod. Den nya pekaren kallas för ”hatt” och beskrivs syntaxmässigt med tecknet ^. Den här pekaren är automatiskt referensräknad och minneshanteras automatiskt. Den här typen av pekare är den enda pekartypen som kan skickas in eller ut ur en C++/CX funktion. Här nedan följer ett exempel på hur den kan användas i en applikation.

```
// StringTest.h

#pragma once

using namespace Platform;

namespace CXPointerExample
{
    public ref class StringTest sealed
    {
    public:
        StringTest();
        String^ HelloString(String^ s);
    };
}

// StringTest.cpp

#include "pch.h"
#include "StringTest.h"

using namespace CXPointerExample;
using namespace Platform;

StringTest::StringTest(){ }

String^ StringTest::HelloString(String^ s)
{
    return "Hello " + s;
}
```

Här har vi definierat en metod som tar in en String-pekare, lägger ihop den med strängen "Hello " och returnerar den nya ihoplagda strängen. Vi ser att en pekare till ett String objekt används både som inparameter och utdata. Det här gör att koden är helt minneshanterat och det är därför vi varken har någon destruktor eller någon kod som friar pekarna eller tar bort objekten vi skapar. Den här koden kan vi nu använda anropa från applikationer i Windows RT.

```
using namespace CXPointerExample;

namespace App1
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
            StringTest stringTest = new StringTest();
            MyTextBox.Text = stringTest.HelloString("World");
        }
    }
}
```

Här har vi en app som innehåller en textbox vars värde vi sätter till det som returneras av HelloString. Vi anropar våra C++/CX funktioner och det här kommer även fungera i andra .NET språk.



Som ni kan se i både applikationen och Windows Runtime komponenten så behöver jag inte hantera minnet på något sätt utan sköts automatiskt av språken.

Att använda WinRT APIet från native C++ kod

I och med att Mario är ett ramverk så måste vissa delar av Mario interageras med det APIet som exponeras på den plattformen den ska köras på. Ett sätt att göra det här är att porta Mario till nått av de språken som WinRT APIet exponeras i men vi vill inte ändra den kodmängden vi redan har. Det skulle kräva en omskrivning av all kod. Visserligen skulle mycket av den kod som redan skrivits kunna köras om man istället kompilerar den som C++/CX kod då det språket delar mycket syntax med C++ men det kräver fortfarande stora omskrivningar. Det vi istället kommer göra är att utreda om det finns tekniker för att anropa WinRT APIet från vanlig C/C++ kod och hur man skulle kunna göra detta när en portning av Mario påböras.

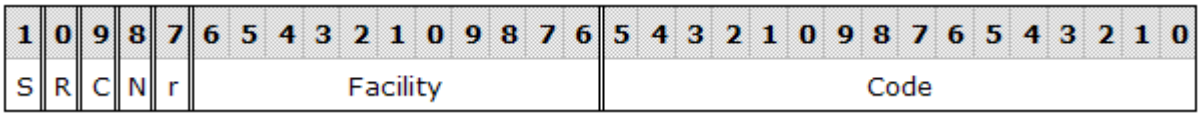
Window Runtime Library

Vi har tidigare pratat om Windows Runtime Librarys och hur de kan användas för att kompilera och köra C/C++ kod på Windows RT. I Windows Runtime Library finns det funktionalitet för att anropa WinRT APIet direkt från C/C++ kod. Det använder en teknik som kallas Component Object Model (COM) för att använda WinRT APIet. COM är en teknik som Microsoft släppte under 1992 och var först och främst tänkt för ett sätt att dela data mellan olika processer. I och med att det är språkoberoende så har det använts mycket väl av Microsoft i många av deras produkter för att kunna skapa en brygga mellan APIer och diverse olika språk.

Det tråkiga med Windows Runtime Library och COM i Windows Runtime är att det inte är en teknik som Microsoft förespråkar även om de erbjuder den. Fokus just nu verkar ligga på att få utvecklare att använda de vanliga .NET språken och C++/CX för att snabbt kunna fylla deras Windows Store med appar för Windows RT. Det här betyder att Windows Runtime Library och hur man använder sig av COM för att anropa Windows RT APIet i dagsläget är helt och hållet odokumenterat och de exempelkoder som finns är otroligt begränsade.[ix]

En viktig fundamental del utav Windows Runtime när man jobbar i C++ är HSTRING. HSTRING är en lågnivå representation av en sträng. Den används av alla språk i Windows RT för att representera strängar men är oftast wrappade av datatyper som är lättare att använda. Den klassiska .NET datatypen System.String är t.ex. en wrappad HSTRING. I Windows Runtime Librarys används HSTRING direkt istället för att wrappas.[x]

Windows Runtime Librarys och Windows Runtime är helt exception fritt. Det här betyder att när du anropar en funktion så kommer den aldrig kasta ett exception om nått går fel och du får inte heller kasta ett exception ut ur ditt bibliotek. För felhantering måste man istället använda HRESULT. Alla funktioner som anropas via C++ från ett Windows Runtime Library returnerar ett HRESULT. HRESULT är ett 32-bitars värde som beskriver om funktionsanropet fungerade eller inte. En HRESULT kan se ut så här.



Biten S beskriver om det uppstod ett fel under anropet eller inte (0 = Success, 1 = Failure).
Bitarna RCNc är reserverade och bör ignoreras.
Facility beskriver vilken komponent felkoden kommer från.
Code är en felkod som beskriver felet.

Det här betyder att varje gång man gör ett anrop in i Windows Runtime så måste man titta på den HRESULT man får tillbaka för att se om den första biten är 0, annars har det uppstått ett fel som måste åtgärdas.[xi]

Windows Runtime Libray har en speciell pekare definierad som heter ComPtr. En ComPtr är en speciell pekare som kan användas för att peka på COM-objekt som man skapar utav olika WinRT klasser. Den här pekaren är referensräknad och minneshantering sker automatiskt.[xii]

```
#include <windows.storage.h>
#include <wrl/client.h>
#include <wrl/wrappers/corewrappers.h>

HRESULT GetTempFolderPath(std::wstring& path)
{
  HRESULT hr;
  Microsoft::WRL::ComPtr<ABI::Windows::Storage::IApplicationDataStat
ics> applicationDataStatics;

  hr =
Windows::Foundation::GetActivationFactory(Microsoft::WRL::Wrappers::
HStringReference(RuntimeClass_Windows_Storage_ApplicationData).Get()
, &applicationDataStatics);
  if (FAILED(hr))
  {
    return hr;
  }

  Microsoft::WRL::ComPtr<ABI::Windows::Storage::IApplicationData>
applicationData;
  hr = applicationDataStatics->get_Current(&applicationData);
  if (FAILED(hr))
  {
    return hr;
  }

  Microsoft::WRL::ComPtr<ABI::Windows::Storage::IStorageFolder>
storageFolder;
  hr = applicationData->get_TemporaryFolder(&storageFolder);
  if (FAILED(hr))
  {
    return hr;
  }

  Microsoft::WRL::ComPtr<ABI::Windows::Storage::IStorageItem>
storageItem;
  hr = storageFolder.As(&storageItem);
  if (FAILED(hr))
  {
    return hr;
  }

  HSTRING folderName;
  hr = storageItem->get_Path(&folderName);
  if (FAILED(hr))
  {
    return hr;
  }
```

```
  UINT32 length;
  PCWSTR value = WindowsGetStringRawBuffer(folderName, &length);
  path = value;
  WindowsDeleteString(folderName);
  return S_OK;
}
```

Här ovan är en simple funktion som hämtar sökvägen till användarens hemkatalog. Här ser vi att ComPtr används för att skapa och peka på objekt från WinRT APIet. HSTRING ges som en ut parameter i `storageItem->get_Path`. Varje gång en funktion har anropats så returneras ett HRESULT och genom att skicka in det i FAILURE funktionen så kan vi avgöra om funktionen fungerade eller inte. Om vi når slutet på funktionen så returnerar vi S_OK som är en för definierad HRESULT som visar att funktionen gick rätt. Det här för att vi ska vara säker på att allt gick bra när vi anropar den här funktionen.

I koden skapar vi en HRESULT och på flera ställen i koden använder vi `if (FAILED(hr))` för att checka om HRESULT värdet ser bra ut. Det här är en av de stora nackdelarna med att Windows Runtime Library är helt utan exceptions. Koden kan köras och vi kan få returvärden ur funktioner även fast saker och ting går fel. I de flesta andra programmeringsspråk så kastas ett exception när nått går fel, men eftersom att den tekniken inte finns tillgänglig måste vi hela tiden titta om de returvärden vi får tillbaka är rätt.

Det FAILED funktionen gör är att titta om S biten i HRESULT är satt till 1 och i så fall returnerar den TRUE. Anledningen till att vi måste göra de här testerna på så många ställe i koden är för att vi måste vara helt säkra på att allt går rätt till när vi anropar en funktion. Om programmet skulle få fortsätta köra trotts att vi hade ett HRESULT som är felaktigt kan vi inte längre garantera vad vi kommer få för resultat.

Koden som är skriven är helt och hållet native C++ kod och visar alltså att det är möjligt att anropa WinRT APIet från vanlig C++ kod. I och med att hela WinRT APIet exponeras på detta sätt så betyder det att Mario är möjligt att porta till Windows RT. Genom att skriva ett Windows Runtime Library för de delar som måste kommas åt av Mario så kan man uppnå detta mål. Den här tekniken skulle också kunna användas av GStreamer för att porta deras plattformsberoende kod till Windows RT.

Mediafångning

Eftersom att Mario är ett ramverk så finns det vissa delar som måste utav det integreras med den plattformen det ska köra på. En av dessa delar är mediafångningen. Vi kommer därför gå igenom hur mediafångnings APIet fungerar och vilka funktioner det har.

I vanliga Windows 8 finns två APIer man kan använda för mediafångning från mediafångningsutrustning (så som kameror och mikrofoner). Det ena är det gamla COM baserade DirectShow APIet som släpptes med Windows 98 eller det nyare Media Foundation som släpptes med Windows Vista. Både DirectShow och Media Foundation kräver att applikationerna du använder de i är skrivna i C++.WinRT APIet har inbyggd funktionalitet för mediafångning. I och med att WinRT APIet själv har hand om mediafångning så kan applikationer som använder sig av detta vara skriva i antingen C++/CX, C#, VB eller JavaScript. I namespacet `Windows.Media.Capture` finns klassen `MediaCapture` som innehåller all funktionalitet som behövs för mediafångning. När man startar en mediafångning kan man ta in datan på tre olika sätt. Antingen kan man skriva en egen `MediaSink`, man kan skriva till en ström eller så kan man skriva till en fil. Det finns även funktionalitet för att använda media i de applikationskontrollerna som är implementerade.

Preview och Record

Alla metoder som finns i `MediaCapture` finns i två upplagor, en för Preview och en för Record (t.ex. `SetPreviewMirroring` och `SetRecordMirroring`). Det är för att en `MediaCapture` har två olika media signaler. Den ena skickar alla frames som videokameran tar emot samt tidsstämplar dem(Record) och en annan som ser till att videon som kommer håller så hög throughput som möjligt genom att droppa gamla frames (Preview). Anledningen till detta är att när man vill spela in video är man inte intresserad av när alla frames når applikationen utan mer intresserad av att den får alla så den kan spara dem. När man däremot vill ha video som är så nära verkligheten som möjligt ska man använda sig av Preview. Metoderna är märkta med Preview då de passar sig bra om man vill rita ut en förhandsvisning på det man spelar in på skärmen. De här metoderna kan även användas i media strömningssyfte då man inte behöver ta hänsyn till eventuell lagg processering av videon.[xiii]

Hur man använder MediaCapture klassen

Jag har skrivit följande kod i C# för att snabbt och enkelt kunna få ihop en prototypapplikation.

```
async private void StartMediaCaputer()
{
    mediaCapture = new MediaCapture();
    await mediaCapture.InitializeAsync();
    myCaptureElement.Source = mediaCapture;
    await mediaCapture.StartPreviewAsync();
}
```

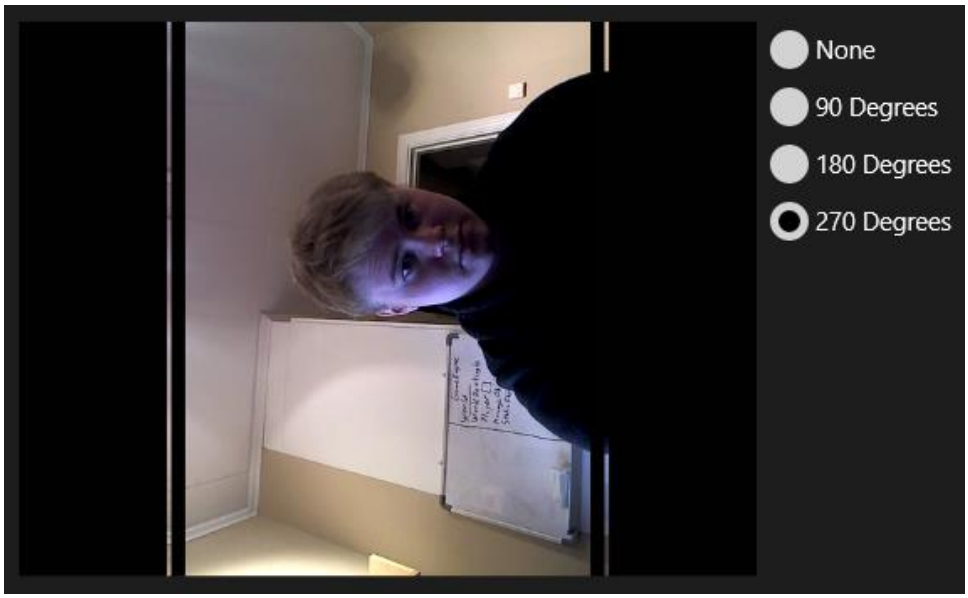
Den här metoden anropar jag från konstruktion och den skapar en `MediaCapture`, initierar den, sätter videokällan för `myCaptireElement` till MediaCapture-instansen jag skapade och startar en preview. De två metoderna `InitalizeAsync` och `StartPreviewAsync` är asynkrona men det är viktigt att de hinner köras klart innan vi går vidare i koden, därför använder vi `await` för att vara säkra på att de blir klara innan vi går vidare. Detta kräver att denna metod måste vara asynkron och har därför taggats med `async` ordet i metodshuvudet.

Buggar i MediaCapture

Jag har stött på en hel del konstiga buggar i MediaCapture klassen. Först och främst så fungerar inte metoden `SetPreviewMirroring(Boolean)` i något annat språk än JavaScript även fast

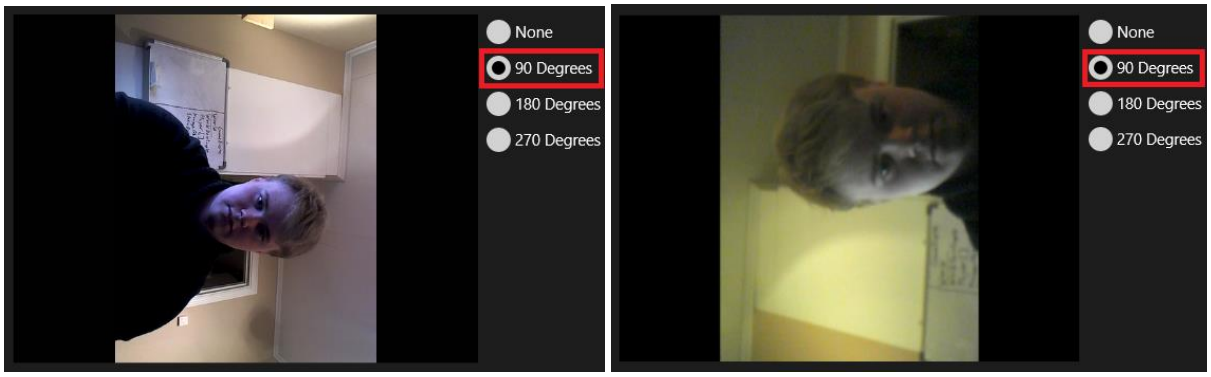
dokumentationen säger att den ska fungera i alla språk som WinRT stöder. Den här metoden ska göra att datan som previewas i `MediaCapture` ska bli spegelvänd.

Om man använder metoden `SetPreviewRotation(VideoRotation)` för att rotera bilden så uppstår det ibland någon typ av störning. Jag har provat detta med två olika kameror för att vara säker på att resultatet är konsistent.



Notera de svarta till höger och vänster i bilden

På min ena kamera (Logitech HD Pro Webcam C920) så fungerar roteringen bra förutom störningen som uppstår vid 270 graders rotering men med min andra kamera (Xbox Live Vision camera) så roteras bilden motsols istället för medsols som den ska göra. Jag har ingen aning vad det här beror på.



Till vänster med Logitech HD Pro Webcam C920 till höger Xbox Live Vision camera.

Jag bör tillägga att några dagar efter jag gjorde upptäckten med rotationsbuggen så släpptes en uppdatering av WinRT SDKn som verkar ha åtgärdat det här problemet.

Som ni ser så är så är det otroligt enkelt att ta in video med hjälp av den nya `MediaCapture` APIt

Hur bör en portning gå till

Eftersom att vi kommit fram till att en portning av Mario inte är möjlig under de premisserna som satts upp så tänkte jag lägga fram lite förslag på hur en portning skulle kunna gå till. Först och främst så måste GStreamer portas eftersom att det används av Mario. Steg ett här är att hitta ett sätt att bygga GStreamer för Windows RT. I nuläget går det att bygga GStreamer med hjälp av GNU Compiler Collection (GCC) vilket ställer till problem eftersom det bara är möjligt att bygga kod för Windows RT med hjälp av Microsoft Visual Studio C++ Compiler (MVSC). Först och främst måste vi skriva någon typ av kompilatorskript för MVSC så att vi kan bygga GStreamer och Mario igenom det. Utöver detta måste även de plattformsspecifika delarna av koden portas för att kunna användas i Windows RT.

När GStreamer går att bygga genom MVSC så måste vi på något sätt brygga ihop det med Windows RT APIet så att vi kan ta in ljud och bild. I och med att GStreamer är pluginbaserat så kan vi ganska lätt få det att fungera på Windows RT. Vi kan då skriva ett plugin som anropar MediaCapture i Windows RT APIet och på så sätt få ett sett att brygga ihop ljud och videoupptagningen med Mario. I och med att GStreamer och Mario är helt skrivna i C så kan vi använda oss av Windows Runtime Library för att i C kod anropa MediaCapture APIet. All funktionalitet som måste nås utav applikationen kan vi wrappa med hjälp av C++/CX metoden för att anropa C++ kod.

I Mario kommer vi behöva skriva om den delen av biblioteket som tar hand om nätverkandet eftersom att Windows RT hanterar nätverk själv i APIet. Att skriva om denna del borde vara trivial eftersom att Windows RT har en socketimplementation som är enkel att använda och har stöd för att skicka bitar mellan hostar med både TCP och UDP.

Mario tar ju även emot ljud och video information och renderar och spelar upp dessa. Vi måste hitta ett sätt för att visa detta i en Windows RT app. I vårt MediaCapture exempel så använde vi oss av ett CaptureElement för att visa videon vi tog in från webkameran. Ett CaptureElement är ett UI element som kan användas för att visa en videoström på skärmen eller spela upp en ljudström i högtalaren. Vi måste skriva en utökning på den här klassen som klarar av att ta den data Mario tar emot och omvandla till något som Windows RT kan rendera och spela upp. Här kan man använda sig av både C++/CX för att wrappa Mario eller Windows Runtime Library, båda teknikerna fungerar och kommer uppfylla samma uppgift.

[i] Google, Bowser på Google Play https://play.google.com/store/apps/details?id=com.ericsson.research.mario&feature=search_result#?t=W251bGwsMSwxLDEsImNvbS5lcmljc3Nvbi5yZXNlYXJjaC5tYXJpbyJd, (2013-01-20)
[ii] Apple, Bowser på App Store https://itunes.apple.com/us/app/bowser/id560478358?ls=1&mt=8, (2013-01-20)
[iii] Microsoft, Windows API references for Windows Store apps (Windows), http://msdn.microsoft.com/en-us/library/windows/apps/br211377.aspx, (2013-01-20)
[iv] Microsoft, Windows Runtime C++ Template Library (WRL), http://msdn.microsoft.com/en-us/library/vstudio/hh438466.aspx, (2013-01-20)
[v] Microsoft, Asynchronous Programming with Asyn and Await (C# and Visual Basic), http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx, (2013-01-20)
[vi] Microsoft, Asynchronous Programming in C++ (Windows Store apps), http://msdn.microsoft.com/en-us/library/windows/apps/hh780559.aspx, (2013-01-20)
[vii] Hans Breuer <hans breuer org>, Re: glib(win32) without libffi, https://mail.gnome.org/archives/gtk-devel-list/2012-July/msg00069.html, (2012-11-02)
[viii] Microsoft, Visual C++ Language Reference (C++/CX), http://msdn.microsoft.com/en-us/library/windows/apps/hh699871.aspx, (2013-01-20)
[ix] Microsoft, Component Object Model (COM), http://msdn.microsoft.com/en-us/library/windows/desktop/ms680573(v=vs.85).aspx, (2013-01-20)
[x] Microsoft, HSTRING, http://msdn.microsoft.com/en-us/library/br205775(v=vs.85).aspx, (2013-01-20)
[xi] Microsoft, HRESULT, http://msdn.microsoft.com/en-us/library/bb446131.aspx, (2013-01-20)
[xii] Microsoft, ComPtr Class, http://msdn.microsoft.com/en-us/library/br244983.aspx, (2013-01-20)
[xiii] Microsoft, MediaCapture class (Windows), http://msdn.microsoft.com/en-us/library/windows/apps/windows.media.capture.mediacapture.aspx, (2013-01-20)