Maarten Struys en Michel Verhagen zijn beiden werkzaam bij PTS Software BV.

Windows CE en het .NET Compact Framework

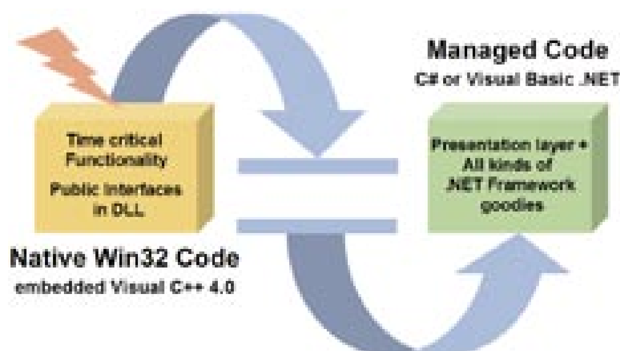
PRODUCTIVITEITSWINST COMBINEREN MET HARD REAL TIME SYSTEEMGEDRAG

Visual Studio.NET 2003 biedt de mogelijkheid om applicaties te ontwikkelen voor zogenaamde smart devices. Dit zijn apparaten die werken met het Windows CE .NET besturingssysteem. De meeste lezers zullen bij dergelijke apparaten denken aan Pocket PC's en Smartphones, maar er is meer. Windows CE is een real time besturingssysteem dat kan worden ingezet voor tal van tijdkritische oplossingen. Zolang rekening wordt gehouden met een aantal randvoorwaarden is het zelfs mogelijk delen van een tijdkritische oplossing te realiseren met behulp van het .NET Compact Framework. In dit artikel kijken we naar de mogelijkheden voor embedded softwareontwikkelaars die gebruik willen maken van een moderne ontwikkelomgeving als Visual Studio.NET 2003.

In de wereld van smart devices is platformonafhankelijkheid met de komst van het .NET Compact Framework een feit. Windows CE kan werken op verschillende hardwareomgevingen. Toch hoeft een .NET Compact Framework-applicatie slechts één keer te worden ontwikkeld. Deze zal dankzij JIT-compilatie zonder verdere aanpassingen werken op X86-, ARM-, MIPS- en SHx-processors. Omdat JIT-compilatie en Garbage Collection schijnbaar tegenstrijdig zijn met hard real time gedrag gaan we hier in dit artikel nader op in en beschrijven we een scenario voor een real time applicatie.

Van unmanaged code naar managed code

In een traditionele ontwikkelomgeving schrijft een ontwikkelaar code in een programmeertaal als C of C++. De bronbestanden worden vervolgens door een compiler vertaald naar machinecode. Deze machinecode kan alleen maar werken op één bepaald type processor. In de pc-wereld is dit niet echt problematisch, omdat nagenoeg alle pc's zijn gebaseerd op Intel X86-processors. Voor smart devices is het verhaal een stuk complexer, omdat bijvoorbeeld Pocket PC's verschillende typen processors kunnen bevatten die niet compatibel zijn met elkaar. De ontwikkelaar moet dan verschillende versies van dezelfde applicatie leveren om alle beschikbare processoren te ondersteunen. In .NET-jargon noemen we dit *unmanaged code* of *native code*; dit betekent dat de code rechtstreeks uitgevoerd wordt op een processor. Met de komst van *managed code* verandert er vooral aan de uitvoerende kant het een en ander. Een ontwikkelaar schrijft nog steeds code in een programmeertaal, maar deze code wordt niet vertaald naar machinecode, maar naar een tussenliggend formaat, in het geval van .NET-applicaties naar Intermediate Language (IL). Pas op het uiteindelijke apparaat wordt IL vertaald naar machinecode door een zogenaamde Just In Time (JIT) compiler. JIT-compilers zorgen ervoor dat managed applicaties door elke ondersteunde processor wordt begrepen. Hiervoor betaalt men wel een prijs. Tijdens uitvoeren van een applicatie vindt de vertaling van IL naar machinecode plaats. De gebruiker moet dus wachten op deze vertaling. Gelukkig gebeurt dit per functie maar éénmalig; eenmaal vertaalde code blijft in het geheugen beschikbaar totdat een applicatie wordt afgesloten of totdat een tekort aan geheugen dreigt.
Naast dit mechanisme van JIT-compilatie wordt een veelvuldige bron van fouten in managed code opgelost. In heel veel applicaties wordt geheugen gebruikt dat nooit meer aan het besturingssysteem wordt teruggegeven, simpelweg doordat de ontwikkelaar dit vergeet te doen. Juist voor embedded toepassingen kan dit voor grote problemen zorgen. Vaak heeft een



Afbeelding 1. Real time scenario, een combinatie van managed en unmanaged

Afbeelding 2. Testopstelling met de single board computer.

embedded systeem een beperkte hoeveelheid geheugen en werken applicaties 7 * 24 uur zonder onderbreking. Als dan geheugen na gebruik niet aan het systeem wordt teruggegeven, is het slechts een kwestie van tijd totdat de vrije hoeveelheid geheugen tot 0 wordt gereduceerd. In een managed omgeving zorgt het systeem ervoor dat niet meer gebruikt geheugen wordt teruggegeven aan het besturingssysteem door een mechanisme wat Garbage Collection (GC) wordt genoemd. Hiervoor betalen we wel een prijs in de vorm van wachttijd terwijl de Garbage Collector aan het werk is. De Garbage Collector onderzoekt regelmatig of stukken geheugen nog in gebruik zijn of dat deze voor hergebruik mogen worden teruggegeven aan het besturingssysteem.

```
using System;
using System.Runtime.InteropServices;

namespace CFinRT
{
   public struct TimingInfo
   {
      public uint AvgIsrIst;
      public uint MaxIsrIst;
      public uint MinIsrIst;
      public uint DeltaMaxIsrIst;
      public uint DeltaMinIsrIst;

      public void Clean ()
      {
         AvgIsrIst = 0;
         MaxIsrIst = 0;
         MinIsrIst = 0;
         DeltaMaxIsrIst = 0;
         DeltaMinIsrIst = 0;
      }
   }

   public class WCEThreadIntf
   {
      [DllImport("RTCF.dll")]
      public static extern bool Init();
      [DllImport("RTCF.dll")]
      public static extern bool DeInit();
      [DllImport("RTCF.Dll")]
      public static extern uint GetTimingInfo(ref TimingInfo ti);
   }

   private TimingInfo  ti;

   public void CollectValue()
   {
      ti.Clean();

      if (WCEThreadIntf.GetTimingInfo(ref ti) != 0)
      {
         ti.MaxIsrIst = (uint)(float)(ti.MaxIsrIst * scaleValue);
         ti.MinIsrIst = (uint)(float)(ti.MinIsrIst *scaleValue);
         ti.AvgIsrIst = (uint)(float)(ti.AvgIsrIst * scaleValue);
         ti.DeltaMaxIsrIst = (uint)(float)(ti.DeltaMaxIsrIst * scaleValue);
         ti.DeltaMinIsrIst = (uint)(float)(ti.DeltaMinIsrIst * scaleValue);
      }

      StoreValue();
      counter = (counter + 1) % samplesInMinute;
   }
}
```

Codevoorbeeld 1.
P/Invoke naar native code

```
// - GetTimingInfo -
// Parameters:
//   lpti -> pointer to a TIMINGINFO structure
// Return value:
//   TRUE on success
RTCF_API BOOL GetTimingInfo(LPTIMINGINFO lpti)
{
   // Tell the IST we need the data now...
   g_bRequestData = TRUE;
   // Wait for the IST to fill the data, but not too long...
   if (WaitForSingleObject(g_hNewDataEvent, 1000)==WAIT_OBJECT_0)
   {
      // Fill the TIMINGINFO structure with the data
      memcpy(lpti, &g_TimingInfo, sizeof(g_TimingInfo));
   }
   else
      return FALSE;
   return TRUE;
}
```

Codevoorbeeld 2.
Native code om timing-gegevens op te halen

Al dit fraais roept echter wel een vraag op. In hoeverre zijn JIT-compilatie en Garbage Collection problematisch voor hard real time gedrag van een applicatie? Windows CE is van oorsprong een hard real time besturingssysteem. Het onvoorspelbare gedrag van JIT-compilatie en Garbage Collection zorgen er jammer genoeg voor dat managed applicaties op zich geen hard real time gedrag vertonen. Zo 'bevriest' de Garbage Collector voor korte tijd alle managed threads in een applicatie als een Garbage Collectie moet plaatsvinden. Als er echter een mogelijkheid zou zijn onderscheid te maken tussen harde real time functionaliteit, geschreven in unmanaged code, en niet real time functionaliteit, geschreven in managed code, rijst de vraag of het beste van twee werelden te combineren valt. Platform Invoke of kortweg P/Invoke is hierop het antwoord.

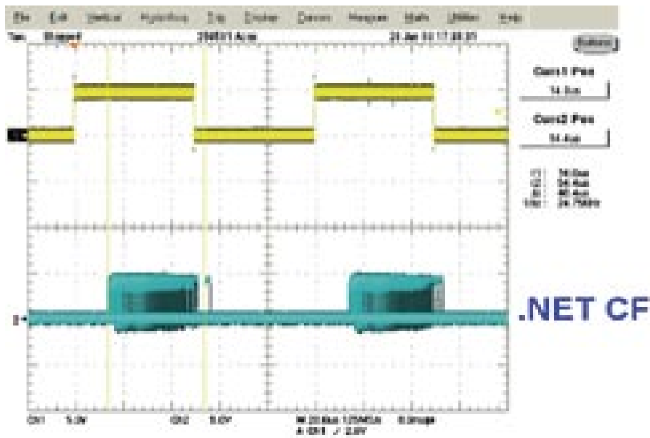
P/Invoke

P/Invoke is de mogelijkheid vanuit een managed applicatie unmanaged functies aan te roepen die ondergebracht zijn in DLL's. Met andere woorden, met P/Invoke kunnen we ontsnappen uit het harnas waarin een managed applicatie zich bevindt. Om van dit mechanisme gebruik te kunnen maken, moeten we een 'kleine schil' in managed code schrijven die een ontsnapping naar een functie in een DLL vormt. In codevoorbeeld 1 is te zien hoe dit in zijn werk gaat. Codevoorbeeld 2 bevat de unmanaged functie die wordt aangeroepen vanuit managed code. Om het .NET Compact Framework in een harde real time toepassing te kunnen gebruiken, passen we dit mechanisme veelvuldig toe. Interessant om te zien is dat de unmanaged functie in codevoorbeeld 2 blokkeert in de API *WaitForSingleObject*, totdat gegevens vanuit een real time Interrupt Service Thread beschikbaar zijn. Doordat we de functie vanuit managed code aanroepen zorgt deze functie er impliciet voor dat onze managed code wordt geblokkeerd.

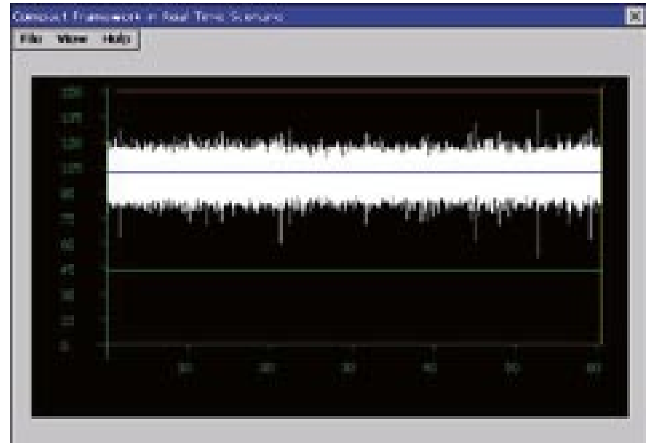
Een scenario voor een real time applicatie

Voor de rest van dit artikel beschrijven we het volgende scenario: een systeem wordt gebruikt om informatie real time van een externe bron op te halen. De ingelezen informatie wordt opgeslagen en uiteindelijk zichtbaar gemaakt voor een gebruiker. Afbeelding 1 laat dit schematisch zien.

Een real time thread, geschreven in C, bevindt zich in een DLL. Elke keer als er een bepaalde interrupt binnenkomt wordt deze thread actief. De interrupt wordt afgehandeld en relevante informatie wordt opgeslagen voor verdere verwerking en voor presentatie. Om gegevens aan een gebruiker te kunnen tonen, wordt een aparte thread gebruikt. Deze is geschreven in C#. Alles wat deze thread doet is informatie



Afbeelding 3. Scoop-afbeelding van managed applicatie: IRQ-IST latency



Afbeelding 4. Schermafbeelding van managed applicatie: IST-starttijden na 100

uitlezen die door de real time thread is opgeslagen en op een grafische manier presenteren aan een gebruiker. Door real time functionaliteit los te koppelen van de userinterface garanderen we dat het systeem real time gedrag vertoond, terwijl de gebruiker toch zo goed mogelijk wordt geïnformeerd. Communicatie tussen de grafische userinterface en de real time delen van het systeem vindt plaats op basis van het P/Invoke-mechanisme dat we eerder hebben beschreven.

Het systeem aan het werk

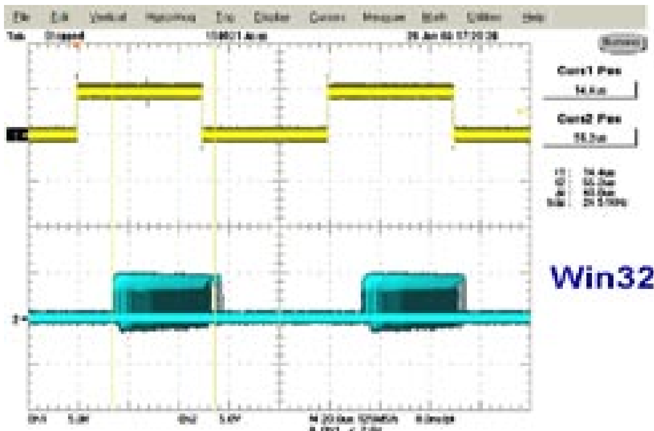
Om te onderzoeken of we het .NET Compact Framework kunnen gebruiken in combinatie met harde real time eisen is een testsituatie nodig die het systeem flink belast, meetbaar is met een oscilloscoop en daarnaast eenvoudig genoeg is om op andere hardware te kunnen worden herhaald. Het systeem moet in staat zijn interrupts te ontvangen en naar aanleiding van die interrupts een signaal naar buiten toe te sturen. Ook worden verschillende responsetijden opgeslagen in het systeem. Voor de test hebben we onze keuze laten vallen op een Single Board Computer die werkt met Windows CE op basis van de Standard SDK; zie afbeelding 2. Dit is een generieke Windows CE-configuratie die een standaard hoeveelheid functionaliteit aan de buitenwereld beschikbaar stelt. In de testopstelling werkt het besturingssysteem op een 300 MHz Geode GX1. Een blokgolf van 10 kHz wordt rechtstreeks aangeboden op IRQ5 van de PC104 bus. Opgaande flanken van de blokgolf genereren interrupts die worden verwerkt door een Interrupt Service Thread (IST). In de IST wordt een puls uitgestuurd naar de parallelle poort van het systeem om ISR-IST latency vervolgens via een scoop zichtbaar te maken. Ook wordt in de IST de huidige tijd opgeslagen. Om metingen over een langere tijd te kunnen waarnemen, slaan we naast de huidige tijd waarop de IST actief wordt ook de minimale, gemiddelde en maximale tijd op die nodig is voordat de IST na aanbieden van een interrupt wordt geactiveerd. Deze informatie wordt op regelmatige tijden doorgegeven aan de grafische userinterface. Gemiddeld zou de IST elke 100 µs actief moeten worden bij een interrupt-frequentie van 10 kHz.
Vanwege JIT-compilatie en Garbage Collection is een volledige managed applicatie niet in staat harde real time eisen waar te maken. Daarom hebben we gekozen voor een opdeling van functionaliteit, waarbij de volledige grafische gebruikersinterface (GUI) in managed

```
// Interrupt Service Thread (unmanaged code)
Wait
On IRQ 5 send probe pulse to the parallel port
Measure time with QueryPerformanceCounter
Store measured time (min, max, current, average) locally
if (userInterfaceRequestsData) {
   copy measured time information
   reset statistic measure values
   set dataReady event
   userInterfaceRequestsData = false
}


// GUI - Periodieke scherm updates (managed code)
disable timer
call with P/Invoke into the DLL
   // Deze code is fysiek terug te vinden in de DLL
   // Dit is dus unmanaged code, aangeroepen vanuit managed code!
userInterfaceRequestsData = true
wait for dataReady event
return measured values
draw measured values on the display, each time using new graphics objects
update marker
enable timer
```

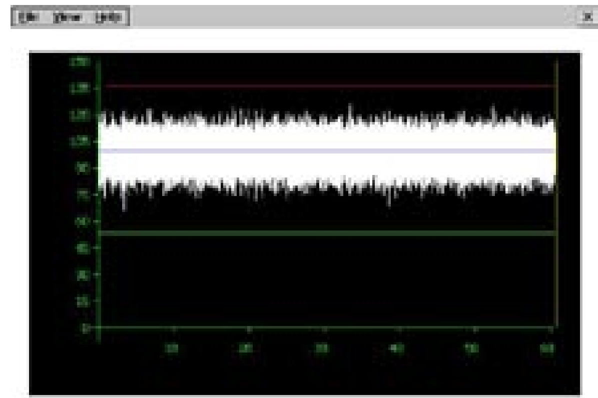
Codevoorbeeld 3.
Het gehele systeem beknopt beschreven in pseudo-code

```
// Op een timer tick halen we nieuwe gegevens op uit de real-time
// thread en tonen we deze op het scherm
private void OnTimer(object source)
{
   // Zet de timer stil tijdens verwerken van een timer tick
   if (theTimer != null)
   {
      theTimer.Change(Timeout.Infinite, dp.Interval);
   }
   Pen blackPen = new Pen(Color.Black);
   Pen yellowPen = new Pen(Color.Yellow);
   Graphics gfx = CreateGraphics();

   td.SetTimePointer(dp.CurrentSample, gfx, blackPen);

   for (int i = 0; i < dp.SamplesPerMeasure; i++)
   {
      td.ShowValue(dp.CurrentSample, dp[i], gfx, i);
   }

   dp.CollectValue();
   td.SetTimePointer(dp.CurrentSample, gfx, yellowPen);

   gfx.Dispose();
   yellowPen.Dispose();
   blackPen.Dispose();

   // Start de timer weer om een volgende timer tick te krijgen
   if (theTimer != null)
   {
      theTimer.Change(dp.Interval, dp.Interval);
   }
}
```

Codevoorbeeld 4.
Periodiek informatie tonen op het scherm



Afbeelding 5. Scoop-afbeelding van unmanaged applicatie: IRQ-IST latency



Afbeelding 6. Schermafbeelding, unmanaged applicatie: IST-starttijden na 100 minuten

code is gerealiseerd. Om communicatie tussen de real time thread en de GUI betrouwbaar te laten werken en het gedrag van het systeem niet te laten beïnvloeden, is gekozen voor dubbele buffering. De GUI vraagt op regelmatige intervallen om nieuwe informatie, maar krijgt deze pas als de real time thread daar tijd voor heeft. Totdat nieuwe data beschikbaar is, blijft de GUI geblokkeerd. Het scherm wordt ongeveer 20 keer per seconde ververst. In codevoorbeeld 3 is te zien hoe het systeem er softwarematig in pseudo-code uitziet.

De resultaten

Tijdens de test hebben we een geheugenscoop aangesloten en na tien minuten een afdruk gemaakt van het scoopbeeld en van het scherm van de applicatie. In afbeelding 3, afkomstig van de scoop, is te zien dat de maximale latency 54.4 µs is en de minimale latency 14.0 µs. De GUI toont een soortgelijk beeld zoals blijkt uit afbeelding 4. In het ideale geval zou de IST elke 100 µs actief moeten worden, wat inderdaad ongeveer de gemiddelde waarde is die uit het systeem komt. Naast het gemiddelde, minimum en maximum per schermupdate bewaren we ook een absoluut minimum en maximum. Het absolute maximum is het slechtste geval dat voorkomt en bepaalt de bruikbaarheid van een systeem. De afwijking gedurende de hele test is beperkt tot ± 40 µs. Dit komt overeen met de scoopwaarneming in afbeelding 3.
De test is een groot aantal keren herhaald; maximaal 100 minuten aan één stuk om er zeker van te zijn dat Garbage Collection en JIT-compilatie veelvuldig aan bod zijn gekomen. Om dit aan te tonen hebben we gebruik gemaakt van een aantal performance-counters die kunnen worden gebruikt om informatie over het gedrag van het .NET Compact Framework te verkrijgen. Deze informatie leverde ook nog eens een extra verificatie op van de juiste werking van het systeem. Ter verificatie hebben we het aantal userinterface-objecten geteld dat tijdens een testduur van 100 minuten dynamisch wordt aangemaakt. Zoals blijkt uit codevoorbeeld 4 worden twee pennen en een grafisch object aangemaakt. In de functies *td.ShowValue* en *td.SetTimerPointer* wordt elk ook nog een grafisch object aangemaakt. Omdat *td.SetTimerPointer* twee keer wordt aangeroepen, worden in totaal zes objecten per schermverversing dynamisch aangemaakt. Dit zijn er 120 per seconde (elke 50 ms wordt het scherm ververst) ofwel 720.000 voor 100 minuten. Uit tabel 1 blijkt dat in werkelijkheid 'slechts' 461.499 objecten zijn aangemaakt, zo'n 35% minder dan verwacht. Door gebruik te maken van de performance-counters wordt volgens Microsoft het systeem ongeveer één derde trager. Dit verklaart het verschil. Het real time gedrag van het systeem werd hier echter niet door beïnvloed. Afbeelding 4 toont dit aan.

Vergelijking met een Win32-applicatie

Om te onderzoeken of het gedrag van een applicatie die managed code en unmanaged code combineert niet afwijkt van het gedrag van een 'traditionele' Windows CE-applicatie, hebben we ook een test uitgevoerd met eenzelfde applicatie, maar dan volledig geschreven in unmanaged code. Deze applicatie maakt gebruik van de zelfde DLL met real time functionaliteit. Tijdens testen blijkt er feitelijk geen verschil te zijn tussen beide applicaties. In afbeelding 5 is te zien dat voor een volledige unmanaged applicatie de maximale latency 55.2 µs is en de minimale latency 14.4 µs. Ook de resultaten, weergegeven door de GUI, zijn nagenoeg identiek zoals blijkt uit afbeelding 6. Mogelijk zitten er kleine verschillen in het aantal schermverversingen per seconde. Vanuit gebruikersoogpunt reageren beide applicaties echter identiek.

De toekomst

Zoals uit dit artikel blijkt is het momenteel noodzakelijk om gebruik te maken van verschillende ontwikkelomgevingen om

Counter	Value	n	Mean	min	max
Total Program Run Time	603752	0	0	0	0
Peak Bytes Allocated	1115238	0	0	0	0
Number Of Objects Allocated	66898	0	0	0	0
Bytes Allocated	1418216	66898	21	8	24020
Number Of Compact Collections	1	0	0	0	0
Bytes Collected By Compact Collections	652420	1	652420	652420	652420
Bytes In Use After Compact Collection	134020	1	134020	134020	134020
Time In Compact Collect	357	1	357	357	357
GC Latency Time	357	1	357	357	357
Bytes Jitted	14046	259	54	1	929
Native Bytes Jitted	70636	259	272	35	3758
Number of Methods Jitted	259	0	0	0	0
Number of Calls	3058607	0	0	0	0
Number of PInvoke Calls	176790	0	0	0	0
Total Bytes In Use After Collection	421462	1	421462	421462	421462

Tabel 1. .NET Compact Framework Performance Counters

enerzijds een hoge productiviteit te waarborgen en anderzijds tijdkritische onderdelen van een applicatie te realiseren. Als Visual Studio 2005 op de markt verschijnt, zal dit niet meer nodig zijn. Met Visual Studio 2005 wordt het mogelijk voor embedded toepassingen zowel managed code in Visual Basic.NET of C# te schrijven en daarnaast native code te schrijven in C of C++ in één en dezelfde ontwikkelomgeving. Omdat het nog een aantal maanden duurt voordat Visual Studio 2005 verschijnt, is het verstandig niet te wachten op deze ontwikkelomgeving, maar vandaag te beginnen met een combinatie van Visual Studio.NET 2003 en embedded Visual C++ 4.0. Met de komst van Visual Studio 2005 zal ontwikkelen voor devices wel eenvoudiger worden, maar de principes zoals beschreven in dit artikel zullen blijven gelden.

Samenwerking door P/Invoke

Het is mogelijk het .NET Compact Framework te gebruiken binnen tijdkritische toepassingen, mits de tijdkritische code zelf in C of C++ wordt geschreven. Dankzij de mogelijkheid van samenwerking tussen managed en native code door middel van P/Invoke, is het mogelijk productiviteitswinst te combineren met hard real time systeemgedrag. Tijdens een groot aantal testen is gebleken dat typisch niet-deterministische systeemdelen zoals JIT en de Garbage Collector geen invloed hebben op tijdkritische code. Op dit moment zijn wij aan het onderzoeken of meer functionaliteit in managed code kan worden ondergebracht met behoud van hard real time gedrag.

Maarten Struys is werkzaam bij PTS Software BV (www.pts.nl), een bedrijf dat zich specialiseert in technische systeemontwikkeling en veel expertise heeft opgebouwd op het gebied van embedded toepassingen met Windows CE en Windows XP Embedded. Maarten is .NET Compact Framework MVP en Windows Embedded Evangelist. Maarten schrijft veelvuldig over het .NET Compact Framework op zijn eigen website www.dotnetfordevices.com.

Michel Verhagen is werkzaam bij PTS Software BV als Windows CE consultant. Michel is embedded MVP. Hij houdt zich vooral bezig met complexe Windows CE-projecten. Daarnaast geeft Michel regelmatig presentaties op internationale conferenties en workshops.

Nuttige URL's

http://msdn.microsoft.com/embedded
http://msdn.microsoft.com/mobile
www.dotnetfordevices.com
www.opennetcf.org
www.windowsfordevices.com
www.microsoft.com/whdc/system/platform/embedded/
http://msdn.microsoft.com/library/en-us/wceintro5/html/wce50conwhatsnew.asp
http://msdn.microsoft.com/library/en-us/dnanchor/html/windowsce.asp