A CROSS-PLATFORM .NET CUSTOM CONTROL ARCHITECTURE FOR HUMAN MACHINE INTERFACE RUNTIME APPLICATION

Yandong Zhong

Master Thesis
System on Chip Design
Royal Institute of Technology

Supervisor: Lingyun Wang
Examiner: Prof. Axel Jantsch

TRITA-ICT-EX-2012:61
Abstract

The Human Machine Interface (HMI) is where interaction between humans and machines occurs. It is widely used in industry such as automatic control field. Controls such as button, text box and so on are fundamental and commonly used in the HMI. Controls typically contain the User Interface (UI) rendering and the business logic. The UI is the graphic display for users, and business logic is logic unit used to handle events of controls in UI. A typical HMI platform usually provides standard controls for users, but users may also want to use .NET custom controls such as self-defined controls and third-party controls. The feature of supporting .NET custom controls is what the thesis is concern on.

Siemens is developing a new HMI platform in which the HMI runtime software has no prior knowledge of a .NET custom control until it is loaded during runtime. However, the business logic of the .NET custom controls will ideally reside in the HMI runtime software, where the full runtime functionalities are available. The key of this concept is to separate the UI rendering of a control and the business logic of a control. Hosting business logic in the runtime will allow sharing the stable and UI-independent business logic across multiple platforms.

In this thesis, a research of existing solutions for the cross-platform custom control concept is carried out, and the concept of .NET custom controls, possible architecture of the cross-process communication model for the thesis topic are defined. The solution of a cross-platform .NET custom control architecture for Siemens HMI runtime software is proposed and the prototype of the solution is implemented under both the Microsoft Windows 32-bit (Win32) and the Microsoft Windows Embedded CE 6.0 (WinCE 6.0) operating system environment. The prototype fulfills the functionality well under both the Win32 and the WinCE 6.0 environment. Furthermore, the prototype efficiency is measured by testing time overhead for method calling crossing process boundaries in the prototype. At last, a conclusion is made and future work is discussed.
Acknowledgements

First of all, I would like to thank my supervisor, Mr. Lingyun Wang, for his unlimited support and encouragement in both my research and personal development. I also would like to acknowledge Prof. Axel Jantsch for providing me the strongest support in thesis writing as my thesis examiner.

This work was supported by Siemens Corporate Research. I am greatly grateful to the Siemens people who I gained the experience and help from, and especially to Mr. Rajesh Gururaj for his help during my Siemens internship.

I also would like to thank all my friends in Sweden, who I have a precious 2 years’ experience with.

Finally, I greatly appreciate my parents and my brother for everything they have done for me.
Contents

Chapter 1 Introduction ........................................................................................................ 1
  1.1 Motivation ............................................................................................................... 1
  1.2 Problem Definition .............................................................................................. 2
  1.3 Thesis Overview ................................................................................................... 2

Chapter 2 Background ...................................................................................................... 5
  2.1 Human Machine Interface ................................................................................... 5
  2.2 Custom Control Definition .................................................................................. 5
  2.3 Microsoft Windows Embedded CE ...................................................................... 6
  2.4 Microsoft .NET Compact Framework ................................................................. 7
  2.5 Component Object Model ................................................................................... 8
  2.6 Managed/Unmanaged Code Interoperability ....................................................... 8

Chapter 3 Architecture Design ....................................................................................... 11
  3.1 Design Requirement ............................................................................................ 11
  3.2 Components Definition ....................................................................................... 12
  3.3 Communication Mechanism ................................................................................ 12
    3.3.1 Introduction ................................................................................................... 12
    3.3.2 Comparison of Interoperability between Managed and Unmanaged Code in .NET Framework 3.5 and .NET CF 3.5 ................................................................. 13
    3.3.3 Communication Mechanism Used in the Architecture ................................. 21
  3.4 Possible Architecture Solutions .......................................................................... 22
    3.4.1 Introduction ................................................................................................... 22
    3.4.2 Solution One .................................................................................................. 22
    3.4.3 Solution Two ................................................................................................. 24
    3.4.4 Solution Three ............................................................................................... 25
    3.4.5 Conclusion ................................................................................................... 27
  3.5 Cross-platform .NET Custom Control Architecture ............................................ 27

Chapter 4 Architecture Implementation ........................................................................ 29
  4.1 Introduction .......................................................................................................... 29
4.2  Components ................................................................................................................. 30
  4.2.1  Out of Process COM Server Component ................................................................. 30
  4.2.2  .NET Client Component .......................................................................................... 32
  4.2.3  Native Runtime Client Component ......................................................................... 34
4.3  Communication Mechanism ............................................................................................ 35
  4.3.1  Communication between .NET Client Component and Out of Process COM Server Component ................................................................................................................. 35
  4.3.2  Reason for Using In-process COM Server DLL ......................................................... 39
  4.3.3  Communication between Native Runtime Client Component and Out of Process COM Server Component ................................................................................................................. 40
  4.3.4  Interfaces .................................................................................................................. 43

Chapter 5  Verification of Architecture Prototype ...................................................................... 49
  5.1  Introduction ................................................................................................................. 49
  5.2  Functionalities Verification .......................................................................................... 49
    5.2.1  Scenario One ............................................................................................................ 49
    5.2.2  Scenario Two .......................................................................................................... 50
    5.2.3  Scenario Three ....................................................................................................... 51
    5.2.4  Scenario Four ......................................................................................................... 52
  5.3  Prototype Efficiency ..................................................................................................... 53
    5.3.1  Overhead Definition ............................................................................................... 53
    5.3.2  Overhead Measurement Tool .................................................................................. 54
    5.3.3  Measuring Overhead ............................................................................................... 55

Chapter 6  Conclusion and Future Work .................................................................................... 59
  6.1  Conclusion .................................................................................................................... 59
  6.2  Future Work .................................................................................................................. 61

References .................................................................................................................................. 62
List of Tables

Table 3.1  Comparison of interoperability between managed and unmanaged code under .NET Compact Framework 3.5 and Full .NET Framework 3.5

Table 3.2  Comparison of interoperability between managed and unmanaged code under .NET Compact Framework 3.5 and Full .NET Framework 3.5

Table 5.1  Time overhead for displaying a small size document

Table 5.2  Constitution of total overhead time for displaying a small size document

Table 5.3  Time overhead for displaying a large size document

Table 5.4  Constitution of total overhead time for displaying a large size document
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Schema of Platform Invoke in the .NET CF 3.5.</td>
</tr>
<tr>
<td>3.2</td>
<td>Schema of Platform Invoke in the .NET Framework 3.5</td>
</tr>
<tr>
<td>3.3</td>
<td>Schema of MessageWindow Class in the .NET CF 3.5</td>
</tr>
<tr>
<td>3.4</td>
<td>Schema of COM Interoperability in the .NET CF 3.5</td>
</tr>
<tr>
<td>3.5</td>
<td>Schema of directly accessing a managed API in the .NET Framework 3.5</td>
</tr>
<tr>
<td>3.6</td>
<td>Schema of exposing a managed API as a COM API in the .NET Framework 3.5</td>
</tr>
<tr>
<td>3.7</td>
<td>Architecture of solution one</td>
</tr>
<tr>
<td>3.8</td>
<td>Architecture of solution two</td>
</tr>
<tr>
<td>3.9</td>
<td>Architecture of solution three</td>
</tr>
<tr>
<td>3.10</td>
<td>The cross-platform .NET custom control architecture for the Siemens HMI runtime application</td>
</tr>
<tr>
<td>4.1</td>
<td>The cross-platform .NET custom control architecture</td>
</tr>
<tr>
<td>4.2</td>
<td>A simple COM server structure</td>
</tr>
<tr>
<td>4.3</td>
<td>Calling into a SimpleCOMServer object from the .NET client component</td>
</tr>
<tr>
<td>4.4</td>
<td>Communication between the .NET client component and the out of process COM server component</td>
</tr>
<tr>
<td>4.5</td>
<td>A Simple COM server structure</td>
</tr>
<tr>
<td>4.6</td>
<td>Communication between the native runtime component and the out of process COM server component</td>
</tr>
<tr>
<td>4.7</td>
<td>Communication between the native runtime client component and the .NET client component using the out of COM server component</td>
</tr>
<tr>
<td>5.1</td>
<td>Method calling path in the test scenario</td>
</tr>
<tr>
<td>6.1</td>
<td>The cross-platform .NET custom control architecture for a Siemens HIM runtime application</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CCW</td>
<td>COM Callable Wrapper</td>
</tr>
<tr>
<td>CLR</td>
<td>Common Language Runtime</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>GUID</td>
<td>Globally Unique Identifier</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface</td>
</tr>
<tr>
<td>MIDL</td>
<td>Microsoft Interface Definition Language</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>RCW</td>
<td>Runtime Callable Wrapper</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SCADA</td>
<td>Supervisory Control And Data Acquisition</td>
</tr>
<tr>
<td>STA</td>
<td>Single-Threaded Apartment</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>Win32</td>
<td>Microsoft Windows 32-bit</td>
</tr>
<tr>
<td>WinCE</td>
<td>Microsoft Windows Embedded CE</td>
</tr>
<tr>
<td>.NET CF</td>
<td>.NET Compact Framework</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 Motivation

SIEMENS is developing a new HMI innovation platform. The goal of the new HMI runtime platform is to harmonize the existing, heterogeneous HMI products ranging from small HMI Panels to sophisticated Supervisory Control And Data Acquisition (SCADA) systems. All these products are going to be built upon the new HMI runtime platform and extended by product-specific functionalities if required.

Controls are fundamental and common in the HMI and typically contain the UI rendering and the business logic. The UI is the graphic display for users, and business logic is logic unit used to handle events of controls in the UI. A typical HMI platform usually provides standard controls for users, but users may also want to use .NET custom controls such as self-defined controls and third-party controls. Therefore, the new HMI platform has the design requirement of supporting .NET custom controls. So, the new platform aims to not only provide plenty of standard controls but also support loading .NET custom controls such as self-defined controls and third-party controls during runtime in the new platform.

Since .NET custom controls are loaded during runtime, the HMI runtime application in the platform has no prior knowledge of a .NET custom control until it is loaded during runtime. However, the business logic of the .NET custom control will ideally reside in the HMI runtime application, where the full runtime functionalities are available. The key of this concept is to separate the UI rendering of a control and the business logic of a control. Hosting business logic in the runtime will allow sharing the stable and UI-independent business logic across multiple platforms.

The final solution of a cross-platform .NET custom controls architecture should support the tight integration (e.g. communication) between the UI rendering and the business
logic, and be generic so that it can be applied to multiple Siemens HMI platforms without much of an effort.

1.2 Problem Definition

The goal of this thesis is to develop a cross-platform .NET custom control architecture for a Siemens HMI runtime application and a prototype to validate the architecture on an embedded Panel device with the WinCE 6.0 operating system environment and a standard Windows PC with the Win32 operating system environment. The prototype shall be designed generically so that it can be deployed and run on multiple Siemens HMI platforms without much of an effort.

1.3 Thesis Overview

This thesis was primarily focused on defining a concept of .NET custom controls, the business logic in the HMI runtime unit and the control logic in the container holding controls, as well as an architecture of the communication model between the business logic and the UI. Then the solution for the cross-platform .NET custom control architecture was determined and the prototype was implemented to verify the solution. In addition, the functionality of the prototype was tested and the prototype efficiency was calculated by measuring time overhead of method calling across components in the architecture.

This thesis is organized as follows.

Chapter 1 provides a brief introduction of the motivation, problem description and the structure of this thesis.

Chapter 2 discusses background information about the HMI, .NET custom controls, the WinCE, the .NET compact framework, the Component Object Model (COM), the interoperability between managed and unmanaged code and so on, which are the
fundamental knowledge used in the .NET custom control architecture design. Literature research of this thesis topic is carried out as thesis references.

Chapter 3 presents the .NET custom control architecture design from components definition, communication model between the business logic and the UI to the whole architecture design. Several possible architecture solutions are proposed and evaluated to find out the final solution.

Chapter 4 implements the prototype of the .NET custom control architecture solution. Components and communication mechanism in the solution are illustrated. The prototype can work successfully both on the Win32 and the WinCE operating system environment, which demonstrates the architecture solution.

Chapter 5 describes prototype test and verification process. Functionalities of the prototype are verified through several verification scenarios. Besides, the prototype efficiency is measured and analyzed by measuring time overhead of method calling across process boundaries in the prototype.

Chapter 6 summarizes the work of this thesis project and discusses further work of this topic.
Chapter 2

Background

2.1 Human Machine Interface

The HMI [1] is the space where interaction between humans and machines occurs and it is usually used in the industry field. Good HMI can make the operator control the machine more effectively and obtain feedback from the machine which helps the operator with making operational decisions [2]. The HMI is widely used in many industry fields including power system, machinery operator controls, automatic controls, electronics device and so on. Especially as the touch screen development is more and more mature, the HMI in electronics device is much more powerful than before.

The HMI includes hardware and software components. Hardware components are the physical part of the HMI which accept input generated by users’ manipulation and display corresponding output indicating the effects of users’ manipulation. Software components are the logical part of the HMI which are used to operate proper logic functions by handling input and producing output. In general, the goal of the HMI is to produce a user interface which is easy and efficient for users to operate a machine in the way which produces the desired result.

2.2 Custom Control Definition

To define the concept of custom controls [3], it is important to clarify the different types of custom controls. In the thesis, custom controls are classified to the following 3 types:

1. The faceplate, which is a composition of controls. Every control in a faceplate is an instance of the control itself. Users may add new properties to the faceplate control.
2. The inherited control, which is a control inherited from an existing control with some custom changes in the properties and

- No change in the UI. In this case, users can add new properties to the new existing-inherited control, and inherit all default properties, events/action handlings from the existing control.

- Some custom changes in the UI. In this case, users can override the default image of the control. Users can provide their own drawing for the control. However, the UI event handlings are still from the existing control.

3. The custom-drawn control, which is a totally new control where the UI and the business logic are created from scratch.

Three types of .NET custom controls are typically not isolated but used together. The following is one example how a user can create a new control. First, an inherited control “custom-button” with a 3-D custom UI can be created from the existing button control. Secondly, another inherited control “custom-io-field” can be created from an existing IO field control and a few new properties can be added. Lastly, some existing static “text field” control, the newly created “custom-button” and “custom-io-field” are grouped together in a faceplate “my control”.

### 2.3 Microsoft Windows Embedded CE

The Microsoft Windows Embedded CE [4] is a real-time operating system developed by Microsoft for embedded systems used in a wide range of consumer and enterprise devices with small memory footprint. The latest version 7 is named as Windows Embedded Compact and previous versions are known as Windows Embedded CE, and they are usually abbreviated as WinCE. The WinCE is a distinct operating system and kernel from desk Windows operating system, which means that the WinCE cannot be seen as a trimmed-down version of the desktop Windows operating system. The WinCE is targeted to devices that have limited hardware resources, such as hand-held devices which may have minimal storage. The WinCE 6.0 is used in this thesis. It is the sixth
major release of Windows Embedded Operating System targeted to enterprise specific tools such as industrial controllers and consumer electronics devices like digital cameras.

2.4 Microsoft .NET Compact Framework

The Microsoft .NET Framework [5] needs to be introduced before discussing the Microsoft .NET Compact Framework. The .NET Framework is a software framework that runs primarily on Microsoft Windows platform. It includes a large framework class and an execution environment known as the Common Language Runtime (CLR). The framework class library provides a number of functionalities to all languages using the .NET Framework. CLR is the core concept of the .NET Framework which provides a language-independent platform for application development and execution. By implementing this important feature, the .NET Framework provides language interoperability across several programming languages.

The Microsoft .NET Compact Framework (.NET CF) [6] is a limited version of the .NET Framework that is designed to run on resource limited embedded devices such as mobile phones, factory controllers, etc. Some class libraries in the .NET CF are the same as in the full .NET Framework, however, there are also a few exclusive libraries in the .NET CF such as the .NET CF controls for mobile devices.

The .NET CF is a subset of the .NET Framework and it is scaled down to use less memory space. Therefore, the .NET CF only supports about 30 percent of the namespaces and classes of the .NET Framework, and is about 8 percent of its size. To keep the size small, only the most important classes and members are supported.

The .NET CF 3.5 is used in this thesis.
2.5 Component Object Model

The Component Object Model (COM) [7] is software architecture developed by Microsoft that allows applications to be built from binary software components. COM objects are discrete components, and each COM object has a unique identity number. COM objects expose interfaces that applications and other components can use to access features defined in the COM object. COM objects are more powerful than Dynamic Link Library (DLL) since they are platform-independent, have built-in inter-process communications capability, and can be easily embedded in object-oriented system.

In the thesis, a single universal COM object is created to provide interfaces for communication between the HMI runtime application and the UI of .NET custom controls.

2.6 Managed/Unmanaged Code Interoperability

The CLR in .NET Framework provides the interoperability [8] of managed code such as C# and other types of unmanaged code such as C++. There are lots of differences on data types, memory management, exception handling mechanisms, security, and design rules between managed and unmanaged object models, but the CLR conceals the differences between them and takes care of the interoperability work.

The interoperability is bidirectional, which makes it possible to call into unmanaged code from managed code and expose managed code to unmanaged code. There are mainly three technologies to achieve the interoperability between managed and unmanaged code.

1. Platform Invoke enables calling from managed code into unmanaged code. Managed code can call a function in unmanaged code if its signature is declared in managed code.

2. COM interoperability enables calling methods in COM components in a manner similar to using normal managed components, and vice versa. So it is allowed to call into native COM object functions from managed code, as well as to call managed functions exposed through COM interfaces from native COM object.
3. C++ interoperability is a C++ specific feature, which enables flat Application Programming Interfaces (APIs) and COM APIs to be used directly, as they have always been used. This is more powerful than COM interoperability, but it also requires much more care.
Chapter 3
Architecture Design

3.1 Design Requirement

The goal of this thesis is to develop a cross-platform .NET custom control architecture for a Siemens HMI runtime application. It is required that the HMI can load not only standard controls provided by Win32 or WinCE operating system, but also .NET custom controls like user-design controls or third-party controls.

Because .NET custom controls are loaded when the HMI runtime application is running, the HMI runtime application has no prior knowledge of .NET custom controls until they are loaded, which means solutions like predefining business logic of specific controls won’t meet the architecture design requirements. Therefore, it is required that a generic business logic of controls must be designed to handle different kinds of events triggered by different controls.

The business logic of controls and the UI rendering of controls are separated. The business logic resides in the Siemens HMI runtime application, while the UI rendering is in a separate process. The key of this concept is to separate the concern: the UI rendering of a control and the business logic of the control. Hosting business logic in the runtime will allow sharing the stable and UI-independent business logic across multiple platforms.

The architecture is required to work on both the Win32 operating system environment and the WinCE operating system environment. Kinds of .NET framework are different in the Win32 and the WinCE operating system environments. The .NET CF in the WinCE operating system environment is a subset of the .NET framework in the Win32 operating system environment, so the architecture must be designed generically so that it can be applied to both platforms without much of an effort. A prototype is needed to be implemented to validate the architecture on an embedded Panel device with the WinCE
6.0 operating system environment and a standard the Windows PC with Win32 operating system environment.

3.2 Components Definition

There are three main components in the architecture: a native runtime client component, a .NET client component which holds .NET custom controls and a COM server component which provide communication interfaces between the native runtime client component and the .NET client component.

The native runtime client component is the HMI runtime application, which is keeping running. The business logic of controls is implemented in the native runtime client component. The business logic of controls should be generic.

The .NET client component is a .NET application which will hold .NET custom controls during runtime. It has no knowledge of controls until they are loaded during runtime. The .NET client component should load .NET custom controls by using DLL files. And the UI rendering work is done in the .NET client component.

The COM server component is designed to provide communication interfaces between the native runtime client component and the .NET client component. It plays a communication bridge role in the architecture.

3.3 Communication Mechanism

3.3.1 Introduction

First, concept of managed code and unmanaged code needs to be introduced. Managed code is the code that executes under the control of the CLR. For example, any code written in C# or Visual Basic .NET belongs to managed code. While the code that runs outside the CLR is unmanaged code. COM components, ActiveX components, and
Win32 API functions are examples of unmanaged code. Unmanaged code is also called native code.

In the architecture design, the native runtime client component is implemented using native code and the .NET client component is implemented using managed code. The business logic of controls resides in the native runtime component and the UI rendering is in the .NET client component, so a proper communication mechanism between managed code and unmanaged code must be established for the architecture.

The architecture is required to work on both the Win32 operating system environment and the WinCE 6.0 operating system environment, so the communication mechanism for the architecture must be available on both the Win32 operating system environment and the WinCE 6.0 operating system environment. The .NET Framework is used in Win32 operating system environment and the .NET CF is used in the WinCE operating system environment, therefore the interoperability between managed and unmanaged Code under the .NET Framework and the .NET CF is investigated. The .NET Framework 3.5 and the .NET CF 3.5 which are used in the architecture are chosen to make a comparison.

3.3.2 Comparison of Interoperability between Managed and Unmanaged Code in .NET Framework 3.5 and .NET CF 3.5

The .NET CF is a subset of the .NET Framework. The .NET CF supports about 30 percent of the namespaces and classes of the .NET Framework, and is about 8 percent of its size. To keep the size small, only the most important classes and members are supported. The .NET CF also contains some features and classes that are specific to mobile and embedded development.

The table below lists some comparison results of interoperability between managed and unmanaged code under the .NET CF 3.5 and the .NET Framework 3.5.
### Evaluation Items

<table>
<thead>
<tr>
<th>Evaluation item</th>
<th>.NET CF 3.5</th>
<th>.NET Framework 3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Invoke</td>
<td>Supported Only one-way function calls from managed code to native code</td>
<td>Supported Bidirectional calls</td>
</tr>
<tr>
<td>MessageWindow Class</td>
<td>Supported Communication from native code to managed code</td>
<td>Not Supported</td>
</tr>
<tr>
<td>COM Interoperability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hosting the runtime from native code</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Using regasm.exe to register a managed component</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Activating managed types from COM</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Hosting ActiveX controls</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Auto generating class interfaces</td>
<td>Not supported You must wrap the functionality you’d like to expose to native code in an explicit interface.</td>
<td>Supported</td>
</tr>
<tr>
<td>Auto generating Globally Unique Identifier (GUID)</td>
<td>Not supported You must give each interface an explicit GUID in order to expose it to native code.</td>
<td>Supported</td>
</tr>
<tr>
<td>C++ Interoperability</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Directly accessing a managed API from native code</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Exposing a managed API as a COM API</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Table 3.1 Comparison of interoperability between managed and unmanaged code under the .NET CF 3.5 and the .NET Framework 3.5

Evaluation Items in the table above will be discussed in the following section. Platform Invoke, MessageWindow Class and COM Interoperability are the main focus of the discussion.

#### 3.3.2.1 Platform Invoke

The .NET CF 3.5: Supported, One-way calls

The .NET Framework 3.5: Supported, Bidirectional calls
Platform Invoke [9] is a feature in .NET framework which is used to call methods in a native DLL from managed code. To call a method in a native DLL from managed code, you need to create a P/Invoke wrapper. A P/Invoke wrapper looks just like a declaration of a managed code method.

Platform Invoke in .NET Compact Framework 3.5 only supports one-way function calls, from managed code to native code, because Asynchronous delegates are not supported.

![Figure 3.1 Schema of Platform Invoke in the .NET CF 3.5](image1)

Platform Invoke in the .NET Framework 3.5 supports bidirectional calls, both from managed code to native code, and callback methods from native code to managed code. Callbacks are supported through the use of delegates which are object oriented function pointers. Delegates pointing to a method in managed code are passed to the function in a native DLL. Then a function in the DLL can call the managed application at the address pointed by the delegate.

![Figure 3.2 Schema of Platform Invoke in the .NET Framework 3.5](image2)

### 3.3.2.2 MessageWindow Class

The .NET CF 3.5: Supported  
The .NET Framework 3.5: Not supported
MessageWindow [9] Class is a way for communicating from native code to managed code. MessageWindow Class is a wrapper around an empty, invisible Win32 window. It is able to send and receive window based messages. It also has a window handle just like ordinary Win32 windows. Using that window handle, native code can transmit a message into managed code by using its traditional ways like either the Win32 PostMessage or the Win32 SendMessage function. Then the WndProc method based on a MessageWindow method is invoked automatically in managed code. Thus, both native code and managed code use its familiar way to establish a communication from native code to managed code.

MessageWindow Class is included in the .NET CF 3.5 but not in the .NET Framework 3.5. In fact, it is one of the features included in the .NET CF 3.5 while not in the .NET Framework.

![Figure 3.3 Schema of MessageWindow Class in the .NET CF 3.5](image)

### 3.3.2.3 COM Interoperability

The .NET CF 3.5: Supported with limitation
The .NET Framework 3.5: Supported

COM Interoperability [10] is a mechanism using existing COM object from managed code and calling a managed method from inside a COM object. There are two key concepts in COM Interoperability: Runtime Callable Wrapper (RCW) and COM Callable Wrapper (CCW).
RCW is a proxy used by CLR to expose native COM objects to managed code and then allow managed code to call functions in native COM objects. RCW appears as an ordinary object to managed code and it marshals call between managed code and native COM objects, so managed code and native COM objects use their familiar way to send and receive data. To use existing native COM objects from managed code, a reference to the native COM object should be added in the managed code and registered. After that, you can use native the COM object, just as you would use a managed object that exists in another assembly.

CCW is a proxy used by CLR to expose managed code to native COM objects and then allow native COM objects to call functions in managed code exposed through native COM interfaces. CCW is used to marshal calls between managed and native code. It’s possible for a native COM object to call back into managed code. A callback interface is defined first inside a native COM object and the implementation of the callback interface is defined in managed code. A reference of this implementation is passed to native COM objects. Then the COM object is able to make a callback to managed code.

Figure 3.4 COM Interoperability in the .NET CF 3.5
There are some restrictions about using native COM objects in the .NET CF 3.5 Comparing to in the .NET Framework 3.5.

1. In the .NET CF 3.5, it is not supported to host the runtime from native code. A delegate or interface of a managed object must be passed to native code if you want to call this managed object from native code. In other words, it is always started out in managed code in order to expose it to native code.

2. In the .NET CF 3.5, it is not supported to use an assembly registration tool like regasm.exe for registering a managed component as a COM class for native COM objects to use.

3. In the .NET CF 3.5, it is not supported to activate managed types from a native COM object.

4. In the .NET CF 3.5, it is not supported to host an ActiveX control from managed code since a managed application cannot act as an ActiveX container. Although .NET CF 3.5 does not support hosting an ActiveX control from managed code, it is possible to implement it by using such features like RCW, CCW, connection points, call backs and access to control handles.

5. In the .NET CF 3.5, it is not supported to auto generate class interfaces. It is required to generate an explicit interface that wraps the functionality to expose to native code.

6. In the .NET CF 3.5, it is not supported to auto generate GUID. It is required to assign an explicit GUID to each interface in order to expose it to native code.

### 3.3.2.4 C++ Interoperability

The .NET CF 3.5: Not supported

The .NET Framework 3.5: Supported

This technique is a C++ specific feature, which allows managed and native code to exist in the same application and even in the same file. The C++ Interoperability feature allows the native APIs are used directly in managed code without having to use the DllImport attribute. This is done by including the header file and linking the import library.
However, this feature is available in a .NET programming language only if it is managed extensions for C++.

C++ Interoperability feature is not available in the .NET CF 3.5 because managed extensions for C++ is not supported in the .NET CF 3.5.

![Figure 3.4 C++ Interoperability in the .NET CF 3.5](image)

### 3.3.2.5 Directly Accessing a Managed API from native code

The .NET CF 3.5: Not supported
The .NET Framework 3.5: Supported

Native code written in C++ can be compiled as a “mixed mode image” in the Visual Studio .NET C++ compiler. After this is done, the native code can directly access any managed API. However, there are some coding rules that apply to accessing managed objects from native code.

This is a feature that is supported in the .NET Framework 3.5 but not in the .NET CF 3.5.

![Figure 3.5 Schema of directly accessing a managed API in the .NET Framework 3.5](image)

### 3.3.2.6 Exposing a Managed API as a COM API

The .NET CF 3.5: Not supported
The .NET Framework 3.5: Supported

Every public managed class can be exposed to native code through COM Interoperability. The CLR in COM Interoperability takes care of all the COM marshaling issues. This
feature must be used carefully because managed and COM object models are very
different, which means some features available managed code may have no equivalent in
COM field and not be usable in COM. Steps to exposing a managed API as a COM API
are as below:

1. Register assemblies with COM objects.
2. Reference a managed API from COM objects.
3. Call a managed API.
4. Deploy an application for COM objects to access.

This is a feature supported in the .NET Framework 3.5 but not in the .NET CF 3.5.
Managed objects cannot be registered as COM objects in the .NET CF 3.5.

![Diagram](image.png)

Figure 3.6 Schema of exposing a managed API as a COM API in the .NET Framework 3.5

### 3.3.2.7 Evaluation Conclusion

Platform Invoke is useful when you need something that is not supported by the .NET
Compact Framework but is supported by calling a function in a Win32 DLL. Platform
Invoke is easy to implement, but the specific function to be called must be defined at the
compilation time. So Platform Invoke is a static call.

MessageWindow Class provides a way to communicate from native code to managed
code in the .NET CF 3.5. By using MessageWindow class, both native code and managed
code use their familiar technique to establish the communication: native code uses
window messages while managed code uses invocation of methods and access to
properties of an object.
COM Interoperability makes the interoperability between native COM objects and managed code easier. Calling native COM objects from managed code is supported well in the .NET CF 3.5. As for calling a function in managed code from native COM objects, there are some restrictions. Because hosting runtime from native code is not supported in the .NET CF 3.5, it is not supported to call into managed code from native COM object directly. However, it is supported to call back into managed code from within a native COM object if the native COM object is hosted by managed code.

C++ Interoperability, directly accessing a managed API from native code and exposing a managed API as a COM API are features that are supported in the .NET Framework 3.5 but not in the .NET CF 3.5. They are not used in the architecture design.

### 3.3.3 Communication Mechanism Used in the Architecture

After learning and comparing interoperability between Managed and Unmanaged Code under the .NET CF 3.5 and the .NET Framework 3.5 which is discussed in section 3.3.2, we need to choose proper communication mechanism for the architecture.

MessageWindow class is mainly used to transfer messages. Although specific handler functions can be defined when a message is received, it has some limitations such as non-string type parameters cannot be transferred. What’s more, MessageWindow class is not support in the .NET framework 3.5, which means it cannot meet the cross-platform requirement of the architecture that the architecture must work on both Win32 and WinCE operating system environment.

C++ Interoperability, Directly accessing a managed API from native code and Exposing a managed API as a COM API are supported in the .NET Framework 3.5 but not in the .NET CF 3.5, which means it cannot meet the cross-platform requirement of the architecture that the architecture must work on both Win32 and WinCE operating system environment.
Platform Invoke can provide communication support from managed code to unmanaged code and COM Interoperability can provide communication support from unmanaged code to managed code. They are supported in both the .NET framework 3.5 and the .NET CF 3.5. Although Platform Invoke and COM Interoperability both have some usage limitation in the .NET CF 3.5, proper combination use of these two communication mechanisms can meet the communication requirement of the architecture.

As a result, Platform Invoke and COM Interoperability techniques are chosen for the communication mechanism of the architecture.

3.4 Possible Architecture Solutions

3.4.1 Introduction

The goal of this thesis is to develop a cross-platform .NET custom control architecture. Because there is no existing solution for our architecture, possible solutions are needed to be proposed, evaluated, and the final solution is chosen from them. Possible solutions are evaluated one by one before the final solution is determined. In the process of finding the final solution, there are mainly three meaningful possible solutions.

3.4.2 Solution One

In this architecture, the native runtime client component and the .NET client component are executable programs in two separate processes. As a result, a bridge component is implemented for the communication between the native runtime client component and the .NET client component. Because of the COM Interoperability in the .NET Framework 3.5 and the .NET CF 3.5, the bridge component is implemented as a COM server. In the architecture design of solution one, the COM server component is implemented as a DLL object to connect the native runtime client component and the .NET client component. DLL is a module containing functions and data that can be used by another module (application or DLL). DLL is so commonly used that it is an
obvious possible way to connect the native runtime component and the .NET client component.

The COM server component is implemented in DLL type, which is an in-process COM server (a program that runs as a service to another program rather than a standalone one). The DLL generated by the COM server component is referred to the native runtime client component and the .NET client component. Because the communication between the native runtime client component and the .NET client component should be bidirectional, the COM server component is required to expose interfaces to the native runtime client component and the .NET client separately.

Figure 3.7 shows the architecture of solution one.

In solution one, both the native runtime client component and the .NET client component loads the DLL object of the COM server component in its own process. Then a critical problem comes out. The native runtime client component and the .NET client component have their own copy of DLL object of the COM server component. The communication channel cannot be established since there are two different DLL objects. In order to fix this problem, The DLL object singleton [11] feature is considered, which ensures only a unique DLL object is created. However, implementing singletons in a DLL can cause some problems. One problem is if your singleton object marked "Apartment" is in a DLL, the two clients create two Single-Threaded Apartment (STA) threads, and each one creates a singleton object. Both STA threads have the same raw pointer to the singleton, allowing them to call into the singleton simultaneously. As a result, it is possible that
accessing to data in the singleton object is not synchronized. Then it is only a matter of time before data is corrupted.

A temporary solution is to set data sharing section in multiple DLL objects like the code segment below:

```c
#pragma data_seg("flag_data")
int count=0;
#pragma data_seg()
#pragma comment(linker,"/SECTION:flag_data,RWS")
```

But it’s too limited to just share some data, the communication between the native runtime client component and the .NET client component includes not only data transaction but also method calling between them.

Because the COM DLL object singleton issue cannot be solved in solution one, an improved solution is proposed to solve this problem.

### 3.4.3 Solution Two

Because solution one has COM DLL object singleton issue, an alternative solution is needed. Solution two is modified based on solution one, which replace the in-process COM server component with an out of process COM server component. As a result, the COM DLL object is changed to a COM executable object. In the out of process COM server component, you can use the following syntax to declare COM object as a singleton:

```c
DECLARE_CLASSFACTORY_SINGLETON()
```

Figure 3.8 shows the architecture of solution two.

![Figure 3.8 Architecture of solution two](image-url)
Solution two solves the singleton issue of the in-process COM server component successfully using an out of process COM server component. By declaring the out of process COM server component as a singleton, the native runtime client component and the .NET client component access the same COM object when they call into the out of COM server component from each side, which means a communication channel between the native runtime client component and the .NET client component can be established through the out of COM server component.

Solution two works on the Win32 operating system environment well, however, an exception about custom data marshaling [12] (transforming the memory representation of an object to a format suitable for storage or transmission) between managed code and native code comes out when it is tested on the WinCE 6.0 operating system environment. In the architecture, methods calling between the .NET client component and the native COM server component needs to take care of marshaling work. Especially, the custom data passed by the methods calling must be marshaled between these two components. This marshaling work is automatically handled by the .NET Framework 3.5 in the Win32 operating system environment while the WinCE 6.0 operating system environment doesn’t do the marshaling work automatically because of limited features supported by the .NET CF 3.5. This marshaling work must be implemented manually in the WinCE 6.0 operating system environment. This is why solution two can work in the Win32 operating system environment but not in the WinCE 6.0 operating system environment. Although it is possible to do the marshaling work manually, it involves lots of trivial work and application readability will not be good.

Due to this marshaling issue between managed code and native code in the WinCE 6.0 operating system environment, an improved solution is proposed to solve this problem.

3.4.4 Solution Three

Solution three is improved from solution two. From solution two it is proved that the communication between the native .NET client component and the out of process COM
server component has some limitations in the WinCE 6.0 operating system environment because of the limitations of the .NET CF 3.5. Due to these limitations the marshaling of custom data types is leading to exception. This is not happening for the in-process COM server component. For this reason, a wrapper is added in our prototype. This wrapper is an in-process COM server DLL object which acts as a bridge between the .NET client component and the out of process COM Server component.

The COM server component is still an out of process COM server since we need a single COM server instance for both the .NET client component and the native runtime client component. In order to solve the marshaling issue existed in solution two, an in-process COM server DLL object is designed as a wrapper around the out of process COM server component. The in-process COM server DLL object is embedded in the .NET client component. The .NET client component communicate with the in-process COM server DLL object directly, and then the in-process COM server DLL object passes the communication to the out of process COM server component. As a result, the custom data marshaling issue between managed code and native code is eliminated.

Figure 3.9 shows the architecture of solution three.

Figure 3.9 Architecture of solution three

Solution three is working on both the Win32 operating system environment and the WinCE 6.0 operating system environment without the exception problem caused by the custom data marshaling issue in solution two.
3.4.5 Conclusion

Three possible solutions for the cross-platform .NET custom control architecture are mainly discussed. Although solution one and solution two both haven’t meet all the design requirements, they are the foundation of solution three, which is the final solution for the cross-platform .NET custom control architecture for a Siemens HMI runtime application. Solution three is developed based on previous solutions – solution one and solution two by continually being improved. In next section, the cross-platform .NET custom control architecture is described in detail.

3.5 Cross-platform .NET Custom Control Architecture

From the discussion of possible architectures in section 3.4, solution three is chosen as the final architecture solution. Figure 3.10 shows the cross-platform .NET custom control architecture for the Siemens HMI runtime application.

![Figure 3.10 The cross-platform .NET custom control architecture for the Siemens HMI runtime application](image)

The architecture contains a native runtime client component, an out of process COM server component, a .NET client component and an in-process COM server DLL wrapper embedded in the .NET client component.

The native runtime client component is the HMI runtime application. The generic business logic of controls is implemented in the native runtime client component.

The .NET client component holds .NET custom controls. It loads .NET custom controls during runtime. The UI rendering work is done in the .NET client component.
The out of process COM server component provides communication interfaces for both the native runtime client component and the .NET client component. It is declared as a singleton object so that the communication channel can be established.

The in-process COM server wrapper is used to eliminate the marshaling work between the .NET client component and the out of process COM server component.

The detail implementation of the architecture is described in next chapter.
Chapter 4
Architecture Implementation

4.1 Introduction

The .NET Framework 3.5 and the .NET CF 3.5 is supported on the Win32 operating system environment and the WinCE 6.0 operating system environment separately. With the rich set of UI controls and their features available in the .NET Framework, it becomes one of the preferred ways for creating and hosting UI components as part of control logic.

The native runtime client component contains generic business logic for controlling all the controls, and the .NET client component contains the UI rendering logic for the controls.

Using .NET Framework as a development framework for control logic means that the UI framework will be based on managed code, and will not be able to directly invoke native runtime APIs. So it is needed to create a bridge component between the native runtime component and the .NET client component. This bridge component will be an out of process COM server object and will be used for communication between the native runtime component and the .NET client component and vice versa.

The figure 4.1 shows the cross-platform .NET custom control architecture.

Fig 4.1: The cross-platform .NET custom control architecture
The architecture should be designed as a cross-platform architecture according to the design requirement, which means the architecture implementation should work both on the Win32 operating system environment and the WinCE 6.0 operating system environment. Because the .NET CF 3.5 on the WinCE 6.0 operating system environment is a limited version of the .NET Framework 3.5 on the Win32 operating system environment, the architecture implementation should first be guaranteed to work under the WinCE 6.0 operating system environment, and then it is verified on the Win32 operating system environment.

The architecture implementation is the same both on the Win32 operating system environment and WinCE 6.0 operating system environment. And the following description is based on WinCE 6.0 operating system environment.

### 4.2 Components

#### 4.2.1 Out of Process COM Server Component

The out of process COM server component is an ATL [13] Smart Device Project for WinCE 6.0. It implements a communication channel between the Native runtime client component and the .NET client component. It exposes three different interfaces.

The first two interfaces are used for callback mechanism, one for the native runtime client component and the other for the .NET client component. The methods which are declared in these two interfaces are not implemented by the out of process COM Server component but are implemented by the native runtime client component and the .NET client component respectively. A reference to the implementation class is then registered with the out of process COM Server component. Once done the out of process COM server component can then invoke the callback methods when needed. The third interface contains methods which the native runtime client component and the .NET client component invoke in order to initiate the communication between each other. Thus the native runtime client component and the .NET client component communicate with the
out of process COM server component by calling the exposed APIs which are implemented in the out of process COM server component itself, and the out of process COM server component communicates with the native runtime client component and the .NET client component using the callback APIs which are implemented in the two client components.

In this case since the two client components will be communicating with each other through the out of process COM Server component, the out of process COM server is designed to run as a single instance server. And since it’s an out of process COM object, it models a singleton. ATL objects normally acquire a class factory by deriving from CComCoClass. This class includes the macro DECLARE_CLASSFACTORY, which declares CComClassFactory as the default class factory. In order to model the singleton, CComClassFactorySingleton is needed to use, for which the DECLARE_CLASSFACTORY_SINGLETON macro is specified in the objects class definition, like the partial code below:

```cpp

class CMyClass : ..., public CComCoClass< ... >
{
    public:
        DECLARE_CLASSFACTORY_SINGLETON(CMyClass)
    ...
};
```

CComClassFactorySingleton derives from CComClassFactory and uses CComObjectGlobal to construct a single object. Each call to the CreateInstance method simply queries this object for an interface pointer. In this way both the native runtime client component and the .NET client component will receive the same interface pointer from the CreateInstance call, thus creating a communication channel.

The WinCE 6.0 operating system environment does not support the automatic marshalling. Therefore, a proxy/stub marshaling code is needed to create to accompany with the out of process COM server component and the two client components and to make sure communication methods are properly marshaled across process boundaries. In the Win32 operating system environment, if only Object Linking and Embedding (OLE) automation compatible data types are used in the communication methods, it is not
necessary to provide the proxy/stub code as the automatic marshalling is always present there (as part of ole32.dll and oleaut32.dll). But if other data types are used, it is still needed to compile the proxy/stub code and register it in the Win32 operating system environment too.

The proxy/stub project is Win32 Smart Device DLL project. It contains all the files that were generated in the out of process COM server component by the Microsoft Interface Definition Language (MIDL) complier. Besides, a .def file is created in the Proxy/Stub project. Content of the .def file is like below.

```
LIBRARY “Proxy/Stub Name”
EXPORTS
    DllGetClassObject        PRIVATE
    DllCanUnloadNow           PRIVATE
    GetProxyDllInfo          PRIVATE
    DllRegisterServer        PRIVATE
    DllUnregisterServer      PRIVATE
```

There are also some configurations to set. One is to add the symbol REGISTER_PROXY_DLL in the Preprocessor Definition property in the proxy/stub project so that DllMain, DllRegisterServer and DllUnregisterServer functions are generated for automatically registering a proxy DLL. Another one is to add the Remote Procedure Call (RPC) library - rpcrt4.lib to the Additional Dependencies property in the proxy/stub project.

At this point, the proxy/stub project is ready to build. The DLL file generated by the project has to be registered on the WinCE 6.0 operating system environment in order to expose the interfaces defined in the out of process COM server component.

### 4.2.2 .NET Client Component

The .NET client component is basically a container to hold all the controls. These controls make up a complex .NET user interface by taking the advantage of the power of the .NET Framework, interact with the native runtime component, and are easily deployed and configured. Using the .NET Framework as rendering framework provides a
rich user experience, and enhance user interface responsiveness by combined with data manipulation mechanisms of the native runtime component.

The .NET client component is a Smart Device Application project for WinCE 6.0. It has support for all the controls included in the .NET CF 3.5, and also supports .NET custom controls. Controls included in the .NET CF 3.5 can be added to the UI from the control toolbox. .NET custom controls are built into separate binaries which will be copied into a proper location before loaded into the .NET client component. Below is the partial code for loading .NET custom controls.

```csharp
using System;
using System.Reflection;

public void LoadAssembly(string strURL, string strClassName)
{
    Assembly a = Assembly.LoadFrom(strURL);
    object o = a.CreateInstance(strClassName);
}
```

The .NET custom control assemblies are dynamically loaded from a URL using the System.Reflection.Assembly class’s LoadFrom function. The exact location of the file must be specified in the URL for the assembly to be loaded.

The UI rendering logic will loop through all the properties and events of each control loaded once controls are loaded into the .NET client component. The properties and events along with the control name and any value associated with it are sent to the native runtime client component, which then are mapped to their respective business logic. Generic interfaces are designed to handle any given property or event in the same manner. When the UI rendering logic encounters an event of a control, it will dynamically bind the event with an event handler, which forwards the control and event information to the native runtime component. In this way UI designer will not have to worry about event handling for controls, instead just build a rich user interface, and all the events are handled by the runtime.
4.2.3 Native Runtime Client Component

The native runtime client component is a Win32 Smart Device Console Project for WinCE 6.0. In the native runtime client component, an implementation of callback interface for COM server calling back to the native runtime client component is defined.

```cpp
class CNativeClientCallBack : public INativeClientCallBack
{
public:
    STDMETHODIMP NotifyEvent(LPTSTR pControlName, LPTSTR pEventType, LPTSTR pValue);
    STDMETHODIMP StoreProperty(LPTSTR pControlName, LPTSTR pPropertyName, LPTSTR pValue);
    STDMETHODIMP StoreEvent(LPTSTR pControlName, LPTSTR pEventType, LPTSTR pValue);
    ...
};
```

Once the native runtime client component starts, it will create an object of the out of process COM server and an object of the callback interface implementation. Then the reference of the callback interface implementation object will be passed to the out of process COM server. After that, the out of process COM server can invoke the callback to the native runtime client component when it is needed.

A business logic unit of all controls exists in the callback interface implementation. It will judge which event of which control loaded in the .NET client component occurs according to the control’s information sent from the .NET client component. When there is a match, it will execute the specific business logic and send the control name, property name and new property value back to the .NET client component to change the property of that control.

Apart from receiving events notification from the .NET client component, the native runtime client component also has a command line feature for user to input command directly to change the property of a certain control in the .NET client component.
4.3 Communication Mechanism

4.3.1 Communication between .NET Client Component and Out of Process COM Server Component

The .NET CF 3.5 supports COM Interoperability, with which interoperating with existing COM objects is achieved. To use existing COM objects from managed code, you can add a reference to the COM object in managed code. References to COM objects can be DLL, executable files, and type library—a binary file that includes information about interfaces, types, and objects that a COM object exposes.

Then an interoperability assembly is created and automatically added as a reference. The interoperability assembly contains managed types that allow you to program against native COM types. The interoperability assembly does not directly communicate with the COM object. Instead the CLR COM Interoperability layer inserts a special proxy between the managed code and the COM object. When interoperating with the COM object, a conversion process is necessary in order to convert argument data between managed code and the COM object. The .NET CF 3.5 supports marshaling of many different managed types both to and from the COM object. Each managed type that is supported has a default marshaling behavior.

A simple COM server is used to explain the COM Interoperability in the architecture, as shown in figure 4.2.
In figure 4.2 the COM object implements a COM server named SimpleCOMServer. It exposes three different interfaces apart from IUnknown interface. These three interfaces are ISimpleServer, IDOTNETClientCallBack and INativeClientCallBack. ISimpleServer interface contains methods used in the COM server. IDOTNETClientCallBack and INativeClientCallBack interfaces are the callback interfaces for the native runtime client component and the .NET client component. The implementations of these two interfaces are in the two client components.

After implementing the SimpleCOMServer object, the .NET client component which is a new C# smart device project is needed to create. In order to communicate with the SimpleCOMServer object, the .NET client component needs to refer to the DLL file of the SimpleCOMServer object that provides a description of the COM object and its interface. After then, an interoperability assembly is created by the compiler so that the .NET client component can access methods of the SimpleCOMServer object. Before being able to use the methods in the SimpleCOMServer object, the following using directives statements need to be provided in the .NET client component.

```csharp
using SimpleCOMServerLib;
using System.Runtime.InteropServices;
```

The "System.Runtime.InteropServices" namespace contains data types to support COM Interoperability. The "SimpleCOMServerLib" namespace contains managed types to expose the SimpleCOMServer object class and interfaces. After the using directives for these namespaces are added, the methods in SimpleCOMServer can be used in the .NET client component. Below is an example of testButton_click event handler in the .NET client component which calls a method in the ISimpleServer interface of the SimpleCOMServer object. Implementing other event handlers is similar. In this scenario a RCW object is instantiated in managed code that wraps a native COM object and then cast to a managed interface describing the native COM interface. During runtime when the function is called and this stub is executed, the arguments are in-marshaled as specified, then the native function is called, and finally the out-marshaling occurs along...
with some runtime post-processing, cleanup, and error handling. If an exception occurs during the in-marshaling process, the native function will not be called.

```csharp
namespace DOTNETClient
{
    public partial class Form1 : Form
    {
        private ISimpleServer objSimpleServer = new SimpleServer();

        public Form1()
        {
            InitializeComponent();
        }

        private void testButton_Click(object sender, EventArgs e)
        {
            objSimpleServer.NotifyControlEvent(testButton.Name.ToString(), "Click", "Main");
        }
    }
}
```

Figure 4.3 illustrates calling into a SimpleCOMServer object from the .NET client.

Since it is possible with the .NET CF 3.5 to pass delegates to native code, you can pass a delegate to an exposed interface, resulting in the possibility for a COM object to call back into managed code. There are three interfaces in SimpleCOMServer - ISimpleServer, IDOTNETClientCallBack and INativeClientCallBack, as discussed before. Here the interfaces IDOTNETClientCallBack and INativeClientCallBack are used to call a method in the .NET client component and the native runtime client component from the.
SimpleCOMServer object. And specifically IDOTNETClientCallBack is used in context with the .NET client component.

In the example here, whenever there is a property change which needs to be updated in the UI, the SimpleCOMServer object is allowed to pass the name of the control, the property which needs to be changed, and the value to which it will be changed to the .NET client component. Before being able to use the callback interface in SimpleCOMServer object, you have to pass a reference to an implementation of the callback interface to SimpleCOMServer.

![Fig 4.4 Communication between the .NET client component and the out of process COM server component](image)

As shown below, the implementation of the IDOTNETClientCallBack interface is provided by the managed ServerCallBack class in the .NET client component.

```csharp
class ServerCallBack : IDOTNETClientCallBack
{
    public void ShowMsg(string pControlName, string pPropertyName, string pValue)
    {
        MessageBox.Show("Change the " + pPropertyName + " Property of the " + pControlName + " Control to " + pValue);
    }
}
```

To use the IDOTNETClientCallBack interface inside the SimpleCOMServer object, you have to pass a reference to its implementation from the .NET client component. The
ISimpleServer interface in the SimpleCOMServer object provides the SetDOTNETClientCallback method for just this purpose, as shown in the code below.

```csharp
public partial class Form1 : Form
{
    private ISimpleServer objSimpleServer = new SimpleServer();
    private ServerCallBack objServerCallBack = new ServerCallBack();

    public Form1()
    {
        InitializeComponent();
        // Use the method of ISimpleServer to pass a callback object to SimpleCOMServer COM object
        objSimpleServer.SetDOTNETClientCallback((IDOTNETClientCallBack)objServerCallBack);
    }
}
```

This code will pass an implementation of the IDOTNETClientCallBack interface to the SimpleCOMServer object, the SimpleCOMServer object then determines when to actually call back into the method provided by the interface.

### 4.3.2 Reason for Using In-process COM Server DLL

The communication between the .NET client component and the out of process COM server using COM interoperability has some limitations of marshaling custom data types in the .NET CF 3.5 as explained in section 3.4.3, which leads to an exception. This is not happening for in-process COM server DLL. In order to solve this exception, an in-process COM server DLL object is added in the architecture. This in-process COM server DLL object is designed as a wrapper around the out of process COM server component and acts as a bridge between the .NET client component and the out of process COM Server component.

The in-process COM server DLL exposes all the interfaces of the out of process COM server component which is needed for communication between the out of process COM server component and the .NET client component. The .NET client component will refer to the in-process COM server DLL to communicate instead of the out of process COM server component, and the in-process COM server DLL will in turn communicate with the out of process COM server component. In this architecture, the ISimpleServer and
IDOTNETClientCallBack interfaces of the out of process COM server component will be exposed by the in-process COM server DLL.

### 4.3.3 Communication between Native Runtime Client Component and Out of Process COM Server Component

Communication between the native runtime client component and the out of process COM server component is COM interoperability. The native runtime client component can call the methods in the out of process COM server component and the out of process COM server component can call back to the native runtime client component. Because both the native runtime client component and the out of process COM server component are native code, when communication between them occurs, no conversion process is needed and argument data can transfer directly between them.

In order to use the out of process COM server object from the native runtime client component, you have to import the type library of the out of process COM server - a binary file including information about interfaces, types, and objects that a COM object exposes. You can use `#import` command to import the type library file (.tlb). The out of process COM server object is still the SimpleCOMServer described in figure 4.2, as shown in figure 4.5 below.

![Fig 4.5 A Simple COM server structure](image-url)
There are three interfaces in SimpleCOMServer - ISimpleServer, IDOTNETClientCallBack and INativeClientCallBack. Here the interfaces IDOTNETClientCallBack and INativeClientCallBack are used to call a method in the .NET client component and the native runtime client component from the SimpleCOMServer object. And specifically INativeClientCallBack is used in context with the native runtime client component, as shown in figure 4.6 below.

Fig 4.6 Communication between the native runtime component and the out of process COM server component

The native runtime client component is a Win32 Smart Device Console Project. Once it is created, the type library file of the COM Server is needed to import.

```cpp
#import "SimpleCOMServer.tlb" named_guids raw_interfaces_only
```

Next an implementation of the callback interface is needed to define in the native runtime client component. As shown below, the implementation of INativeClientCallBack interface is defined in the CNativeClientCallBack class.

```cpp
class CNativeClientCallBack : public INativeClientCallBack
{
public:
    STDMETHODIMP ShowMsg(LPTSTR pControlName, LPTSTR pEventName, LPTSTR pValue);
    STDMETHODIMP StoreProperty(LPTSTR pControlName, LPTSTR pEventName, LPTSTR pValue);
    STDMETHODIMP StoreEvent(LPTSTR pControlName, LPTSTR pEventName, LPTSTR pValue);

public:
    CNativeClientCallBack(ISimpleServer* pSimpleServer);
    ~CNativeClientCallBack(void);
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG) AddRef();
    STDMETHODIMP_(ULONG) Release();
private:
    ULONG m_nRefCount;
    ISimpleServer* m_pSimpleServerCopy;
};
```
To use the INativeClientCallBack interface inside the SimpleCOMServer object, a reference to its implementation is needed to pass from the native runtime client component. The ISimpleServer interface in the SimpleCOMServer object provides the SetNativeClientCallback method for just this purpose, as shown in the code below.

```csharp
int _tmain(int argc, _TCHAR* argv[])
{
    CNativeClientCallBack *m_pNativeClientCallBack;
    ISimpleServer* m_pSimpleServer;
    CoCreateInstance(CLSID_SimpleServer,NULL,CLSCTX_LOCAL_SERVER,__uuidof(m_pSimpleServer),(void**)&m_pSimpleServer);
    m_pNativeClientCallBack = new CNativeClientCallBack(m_pSimpleServer);
    m_pSimpleServer->SetNativeClientCallback((INativeClientCallBack *)m_pNativeClientCallBack);
    ...
}
```

This code will pass an implementation of the INativeClientCallBack interface to the SimpleCOMServer object, the SimpleCOMServer object then determines when to actually call back into the method provided by the interface.

The complete communication between the native runtime component and the .NET client component using the out of COM server component is shown as the figure 4.7 below.

![Communication diagram](image-url)
4.3.4 Interfaces

The ISimpleServer, IDOTNETClientCallBack and INativeClientCallBack interfaces exposed by the out of process COM server component will be used to communicate between the native runtime client component and the .NET client component. Following are the descriptions of the methods in these three interfaces.

4.3.4.1 ISimpleServer

StoreControlEvent
When the controls are loaded by the .NET client component, it will loop through all the events of the loaded controls. If the loaded control is a faceplate, all the sub controls will also be scanned for available events. Each event along with the control name and any value associated with it, will be sent to the native runtime client component using StoreControlEvent(). Once received by the native runtime client component, the event information will be mapped to its respective business logic.

```c
HRESULT StoreControlEvent([in] LPTSTR pControlName, [in] LPTSTR pEventType, [in] LPTSTR pValue);
```

StoreControlProperty
When the controls are loaded by the .NET client component, it will loop through all the properties of the loaded controls. If the loaded control is a faceplate, all the sub controls will also be scanned for its properties. Each property along with the control name and any value associated with it, will be sent to the native runtime client component using StoreControlProperty(). Once received by the native runtime client component, the properties information will be stored.

```c
HRESULT StoreControlProperty([in] LPTSTR pControlName, [in] LPTSTR pPropertyName, [in] LPTSTR pValue);
```

NotifyControlEvent
All the events which can be invoked from the loaded controls will be dynamically mapped to a common event handler, which will extract the event information, the information of the control whose event is fired, and information on any values associated with the fired event. The event handler will then pass this information to the native
runtime client component using NotifyControlEvent(). The business logic for handling the events will be part of the native runtime client component.

```cpp
HRESULT NotifyControlEvent([in] LPTSTR pControlName, [in] LPTSTR pEventType, [in] LPTSTR pValue);
```

**ChangeControlProperty**
The native runtime client component contains all the information about the properties and their values for every control loaded by the .NET client component. If a particular property of a control needs to be changed in the UI, the native runtime client component can notify that change to the .NET client component, using ChangeControlProperty(). The control name, the property name and the value to be changed, will be sent to the .NET client component, so as to apply the change in the UI.

```cpp
HRESULT ChangeControlProperty([in] LPTSTR pControlName, [in] LPTSTR pPropertyName, [in] LPTSTR pValue);
```

**GetPropertyValue**
When the native runtime client component needs the value of a particular property of any control in the UI, GetPropertyValue() will be used to retrieve it. This method is designed because the native runtime client component needs to update the properties of the controls in its database after controls’ properties are changed. The native runtime client component will request the value by specifying the name of the control and the name of the property which needs to be updated.

```cpp
HRESULT GetPropertyValue([in] LPTSTR pControlName, [in] LPTSTR pPropertyName);
```

**SetDOTNETClientCallback**
The .NET client component and the native runtime client component will communicate to each other through the out of process COM server component. This communication will be facilitated through direct method calls and method call backs. The .NET client component can directly call functions in the out of process COM server component, but the out of process COM server component needs to use the callback interface registered by the .NET client component to communicate back. The callback interface is defined in the out of process COM server but implemented in the .NET client component.
SetDOTNETClientCallback() will be used by the .NET client component to register the callback interface to the out of process COM server component.

```
HRESULT SetDOTNETClientCallback([in] IDOTNETClientCallBack* pDOTNETClientCallBack);
```

**SetNativeClientCallback**
The native runtime client component and the .NET client component will communicate to each other through the out of process COM server component. This communication will be facilitated through direct function calls and function call backs. The native runtime client component will directly call functions in the out of process COM server component, but the out of process COM server component needs to use the callback interface registered by the native runtime client component to communicate back. The callback interface is defined in the out of process COM server component but implemented in the native runtime client component. SetNativeClientCallback() will be used by native runtime client component to register the callback interface to the out of process COM server component.

```
HRESULT SetNativeClientCallback([in] INativeClientCallBack* pNativeClientCallBack);
```

**ResetDOTNETClientCallback**
ResetDOTNETClientCallback() will be used by the .NET client component to unregister any callback interfaces registered to the out of process COM server component. Calling this method will disable any communication from the out of process COM server component to the .NET client component.

```
HRESULT ResetDOTNETClientCallback(void);
```

**ResetNativeClientCallback**
ResetNativeClientCallback() will be used by the native runtime client component to unregister any callback interfaces registered to the out of process COM server component. Calling this method will disable any communication from COM server to the native runtime client component.

```
HRESULT ResetNativeClientCallback(void);
```

**4.3.4.2 IDOTNETClientCallBack**
**ChangeProperty**

ChangeControlProperty() called from the native runtime client component to the out of process COM server will be forwarded by the out of process COM server component to the .NET client component using the ChangeProperty() callback method. A reference to the implementation of IDOTNETClientCallBack interface needs to be registered first by the .NET client component in the out of process COM Server component.

```c
HRESULT ChangeProperty([in] LPTSTR pControlName, [in] LPTSTR pPropertyName, [in] LPTSTR pValue);
```

**GetPropertyValues**

When the native runtime client component requests the .NET client component for the value of a particular property in a control, it will call GetPropertyValue() method in the out of process COM server component, which will be forwarded by the out of process COM server component to the .NET client component using the GetPropertyValues() callback method. A reference to the implementation of IDOTNETClientCallBack interface needs to be registered first by the .NET client component in the out of process COM Server component.

```c
HRESULT GetPropertyValues([in] LPTSTR pControlName, [in] LPTSTR pPropertyName);
```

**4.3.4.3 INativeClientCallBack**

**NotifyEvent**

When an event is fired from a control loaded in the .NET client component, the .NET client component will call the NotifyControlEvent() method in the out of process COM server component with the complete event information. The out of process COM server component will then forward the information to the native runtime client component using the NotifyEvent() callback method. The implementation of NotifyEvent() is in native runtime component and contains the business logic for handling the event. A reference to the implementation of INativeClientCallBack interface needs to be registered first by the native runtime client component in the out of process COM Server component.

```c
HRESULT NotifyEvent([in] LPTSTR pControlName, [in] LPTSTR pEventType, [in] LPTSTR pValue);
```
**StoreProperty**
When a control is loaded in the .NET client component, the .NET client component will loop through all the properties of that control and its sub controls and will call StoreControlProperty() method in the out of process COM server component with the properties information. The out of process COM server component will then forward the information to the native runtime client component using the StoreProperty() callback method, and the properties will be stored in the out of process COM server component. A reference to the implementation of INativeClientCallBack interface needs to be registered first by the native runtime client component in the out of process COM Server component.

```csharp
HRESULT StoreProperty([in] LPTSTR pControlName, [in] LPTSTR pEventType, [in] LPTSTR pValue);
```

**StoreEvent**
When a control is loaded in the .NET client component, the .NET client component will loop through all the events which can be fired using the loaded control and its sub controls and will call StoreControlEvent() method in the out of process COM server component with the properties information. The out of process COM server component will then forward the information to the native runtime client component using the StoreEvent() callback method, and the events will be mapped to their respective business logic. A reference to the implementation of INativeClientCallBack interface needs to be registered first by the native runtime client component in the out of process COM Server component.

```csharp
HRESULT StoreEvent([in] LPTSTR pControlName, [in] LPTSTR pEventType, [in] LPTSTR pValue);
```
Chapter 5
Verification of Architecture Prototype

5.1 Introduction
A prototype is developed to implement the cross-platform .NET custom control architecture discussed in chapter four. In this chapter, verification [13] of functionalities of the prototype is introduced. Then the prototype efficiency is tested by measuring the time overhead ratio for method calling in the prototype.

5.2 Functionalities Verification
The cross-platform .NET custom control architecture should implement bidirectional communication between the native runtime client component and the .NET client component. For example, when a .NET custom control is triggered by the user, the event should be transferred to the native runtime component through the out of process COM server component. After the business logic of controls in the native runtime client component receives the event, it should match corresponding handling logic for this event, and pass back correct handling information to UI rendering in the .NET client component so that UI rendering can do the right rendering work. Four scenarios are designed to verify the complete functionalities in the prototype in the Win32 operating system environment and the WinCE 6.0 operating system environment. Tests of these four scenarios are passed and the functionalities of the architecture are verified.

5.2.1 Scenario One
Scenario one is to print out all properties’ and events’ name of the loaded control. When a custom control is loaded in the .NET client component, all properties’ and events’ name of the loaded control will be printed out on the console window of the native runtime client component.
Following is how this scenario works in the architecture:

1. When a .NET custom control is loaded, the .NET client component will call the SendAllEvents() method and the SendAllProperties() method.
2. In the SendAllEvents() method and the SendAllProperties() method, the .NET client component will look for every event/property of the loaded control and pass the control and the event/property name information to the out of process COM server component by calling the StoreControlEvent() method and the StoreControlProperty() method.
3. Then the out of process COM server component will call the StoreEvent() method and the StoreProperty() method defined in the INativeClientCallBack interface implementation in the native runtime client component to display the name information on the console window.

5.2.2 Scenario Two

Scenario two is to change the background color of the button_Click button in the ClickCountingButton control when the button_Click button is clicked. When the button_Click button is clicked, the background color of the button_Click button in the ClickCountingButton control will be changed to red.

Following is how this scenario works in the architecture:

1. All the events which can be invoked from the controls loaded in the .NET client component will be dynamically mapped to the Common_Event_Handler() method.
2. When the button_Click button is clicked, the .NET client component will call the Common_Event_Handler() method.
3. In the Common_Event_Handler() method, the .NET client component will pass the name of button_Click button and the name of click event to the out of process COM server component by calling the NotifyControlEvent() method.
4. Then the out of process COM server component will call the NotifyEvent() method defined in the INativeClientCallBack interface implementation in the...
native client component to pass those information. Business logic for handling control events is defined in the NotifyEvent() method.

5. The business logic for handling the click event of the button_Click button will pass the name of the button_Click button, the name of the BackColor property and the name of color to be changed to the out of process COM server component by calling the ChangeControlProperty() method.

6. Then the out of process COM server component will call the ChangeProperty() method defined in the IDOTNETClientCallBack interface implementation in the .NET client component to change the background color of the button_Click button in the ClickCountingButton control.

### 5.2.3 Scenario Three

Scenario three is to fill in the textBox_FirstName textBox and the textBox_LastName textBox in the NameRegistrationForm control from the console window of the native runtime client component using command line. Users can fill in the textBox_FirstName textBox and the textBox_LastName textBox in the NameRegistrationForm control from the console window of the native runtime client component.

Following is how this scenario works in the architecture:

1. In the console window of the native runtime client component, press F1 to input commands. Input the control name, the property name and the new property value to be changed according to prompt message. In scenario three, those are the names of the textBox_FirstName textBox and the textBox_LastName textbox, the name of the Text property and text to fill in.

2. When input process is finished, the native runtime client component will pass the input values to the out of process COM server component by calling the ChangeControlProperty() method.

3. Then the out of process COM server component will call the ChangeProperty() method defined in IDOTNETClientCallBack interface implementation in
the .NET client component to fill in the textBox_FirstName textBox and the textBox_LastName textBox in the NameRegistrationForm control.

5.2.4 Scenario Four

Scenario four is to display text of the textBox_FirstName textBox and the textBox_LastName textBox in the NameRegistrationForm control in the console window of the native runtime client component. When the button_ViewMessage button is clicked, text of the textBox_FirstName textBox and the textBox_LastName textBox in the NameRegistrationForm control will be displayed in the console window of the native runtime client component.

Following is how this scenario works in the architecture:

1. When the button_ViewMessage button is clicked, the .NET client component will call the Common_Event_Handler() method.
2. In the Common_Event_Handler() method, the .NET client component will pass the name of the button_ViewMessage button and the name of click event to the out of process COM server component by calling the NotifyControlEvent() method.
3. Then the out of process COM server component will call the NotifyEvent() method defined in the INativeClientCallBack interface implementation in the native runtime client component to pass those information. Business logic for handling control events is defined in the NotifyEvent() method.
4. The business logic for handling click event of the button_ViewMessage button will pass the names of the textBox_FirstName textBox and the textBox_LastName textBox, and the name of Text property to the out of process COM server component by calling the GetPropertyValue() method.
5. Then the out of process COM server component will call the GetPropertyValues() method defined in the IDOTNETClientCallBack interface implementation in the .NET client component to get the text content of the textBox_FirstName textBox and the textBox_LastName textBox.
6. After getting the text content, the .NET client component will pass the names of the textBox_FirstName textBox and the textBox_LastName textBox, the name of Text property and the text content of those two textBoxes to the out of process COM server component by calling the StoreControlProperty() method.

7. Then out of process COM server component will call the StoreProperty() method defined in the INativeClientCallBack interface implementation in the native runtime client component to display the text content of the textBox_FirstName textBox and the textBox_LastName textBox in the NameRegistrationForm control in the console window of the native runtime client component.

5.3 Prototype Efficiency

The time overhead for methods calling between the native runtime client component, the out of process COM server component and the .NET client component in the prototype is measured to test the efficiency of the prototype. The ChangeControlProperty() method calling is taken as the measuring scenario: When a document number in a drop list control is selected, the specific document content is displayed in the text box control in the .NET client component.

Measuring test is repeated six times to get the average value. The total time for the measuring scenario, the total overhead time, and the constitution of the total overhead time will be measured. The time overhead under the WinCE 6.0 operating system environment and the Win32 operating system environment is measured. Following time efficiency results are under the WinCE 6.0 operating system environment. The results on the Win32 operating system environment is similar to that in the WinCE 6.0 operating system environment, but with smaller time overhead and time overhead ratio.

5.3.1 Overhead Definition

When the native runtime client component receive the event of the drop list control to display a document content, it will call the ChangeControlProperty() method in the out of
process COM server component, and the out of process COM server component will forward this call to the in-process COM server wrapper, then to the .NET client component. The .NET client component will create a separate thread to execute the method of display specific document content.

Fig 5.1 Method calling path in the test scenario

The time used for initializing instance and passing parameters between the native runtime client component, the out of process COM server component and the .NET client component constitutes the total communication overhead.

### 5.3.2 Overhead Measurement Tool

High Resolution Timer is used at the .NET client component for recording the SetValue time. This timer provides more precise measurements than you can obtain by using the TickCount property, which has 1-millisecond resolution. And for the native runtime client component, QueryPerformanceCounter system method is used.

We declared a start timer and an end timer in the call functions for each component. Following the logical sequence, the receiver’s start timer minus caller’s start timer is one part of communication time.

From the native client component to the out of COM server component

```c++
QueryPerformanceCounter(&strt_ChangeControlProperty);   //start timestamp
m_pSimpleServerCopy->ChangeControlProperty(pControlName, pEventType, pValue);
QueryPerformanceCounter(&ed_ChangeControlProperty);    //end timestamp
```
From the out of COM server component to the .NET client component

<table>
<thead>
<tr>
<th>Iterance</th>
<th>Overhead</th>
<th>Total Time</th>
<th>Display File Time</th>
<th>Overhead Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.709</td>
<td>50.930</td>
<td>43.221</td>
<td>15.1%</td>
</tr>
<tr>
<td>2</td>
<td>3.461</td>
<td>27.203</td>
<td>23.742</td>
<td>12.7%</td>
</tr>
<tr>
<td>3</td>
<td>3.448</td>
<td>27.171</td>
<td>23.723</td>
<td>12.7%</td>
</tr>
<tr>
<td>4</td>
<td>3.476</td>
<td>27.317</td>
<td>23.841</td>
<td>12.7%</td>
</tr>
<tr>
<td>5</td>
<td>3.450</td>
<td>27.091</td>
<td>23.641</td>
<td>12.7%</td>
</tr>
<tr>
<td>6</td>
<td>3.556</td>
<td>27.342</td>
<td>23.786</td>
<td>13.0%</td>
</tr>
<tr>
<td>Average</td>
<td>3.478</td>
<td>27.225</td>
<td>23.747</td>
<td>12.8%</td>
</tr>
</tbody>
</table>

Table 5.2 Time overhead for displaying a small size document

The SetValue time in the .NET client component

5.3.3 Measuring Overhead

The scenario for measuring overhead is like this. A document number from the drop list control is selected, and then the .NET client component will send this number to the native runtime client component. After receiving the number, the native runtime client component returns the specific document name according to the document number to the .NET client component. The .NET client component will display the content of the document in the text box control. In order to test how the control complexity affects time overhead, a small and a large size documents are used to test to compare the time overhead. Below are the measuring results for displaying a small (3 kilobyte) size document. The time unit is millisecond.
<table>
<thead>
<tr>
<th>Iterance</th>
<th>Direction</th>
<th>Native Runtime Component and COM</th>
<th>within COM</th>
<th>COM and COM Bridge DLL</th>
<th>COM Bridge DLL and .NET Client</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Call</td>
<td>0.786</td>
<td>0.013</td>
<td>4.184</td>
<td>1.040</td>
<td>7.709</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>0.910</td>
<td>0.013</td>
<td>0.624</td>
<td>0.139</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Call</td>
<td>0.757</td>
<td>0.013</td>
<td>0.863</td>
<td>0.265</td>
<td>3.461</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>0.873</td>
<td>0.012</td>
<td>0.557</td>
<td>0.121</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Call</td>
<td>0.768</td>
<td>0.012</td>
<td>0.855</td>
<td>0.262</td>
<td>3.448</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>0.861</td>
<td>0.012</td>
<td>0.558</td>
<td>0.120</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Call</td>
<td>0.778</td>
<td>0.013</td>
<td>0.857</td>
<td>0.271</td>
<td>3.476</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>0.859</td>
<td>0.012</td>
<td>0.565</td>
<td>0.121</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Call</td>
<td>0.764</td>
<td>0.012</td>
<td>0.849</td>
<td>0.270</td>
<td>3.450</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>0.872</td>
<td>0.012</td>
<td>0.550</td>
<td>0.121</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Call</td>
<td>0.786</td>
<td>0.014</td>
<td>0.911</td>
<td>0.270</td>
<td>3.556</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>0.875</td>
<td>0.012</td>
<td>0.556</td>
<td>0.132</td>
<td></td>
</tr>
</tbody>
</table>

*Average 3.478

Table 5.3 Constitution of total overhead time for displaying a small size document

*Average Calculation does not take the 1st result into account.

From the table 5.2, it is seen that the communication time in the first time test is 7.709 milliseconds and the time overhead ratio is 15.1%, which takes more time than the following test results. This is because some initial work is needed to do at the first time test. After that, the time overhead stays stable like 3.4 milliseconds and the time ratio stays overhead like 12.8%.

From the table 5.3, it is seen that the communications between the native runtime client component and the out of process COM server component, the out of process COM server component and the in-process COM Bridge DLL are critical. This is because these communications cross different processes and need to take care of marshaling issue, which costs some time.

In order to compare time overhead of displaying different size documents, we test time overhead with a large document. Below are the measuring results for displaying a large (12 kilobyte) size document.
From table 5.4, it is seen that the communication time in the first time test is 7.888 milliseconds and the time overhead ratio is 10.6%, which takes more time than the following test results. This is because some initial work is needed to do at the first time test. After that, the time overhead stays stable like 3.4 milliseconds and the time overhead ratio stays like 7.1%.

From table 5.5, it is seen that the communications between the native runtime client component and the out of process COM server component, the out of process COM server component and the in-process COM Bridge DLL are critical. This is because these
communications cross different processes and need to take care of marshaling issue, which costs some time.

By comparing the time overhead results of displaying a small document and a large size document, it is concluded that the time overhead is fixed no matter how complex the control is. The more complex the control is, the more time used to handle the event is, so that the time overhead ratio is smaller.
Chapter 6
Conclusion and Future Work

6.1 Conclusion

In this thesis, a cross-platform .NET custom control architecture for a Siemens HMI runtime application is developed, and a prototype of this architecture is implemented. This architecture is successfully tested on the Win32 operating system environment and the WinCE 6.0 operating system environment to demonstrate the functionality and efficiency.

The cross-platform .NET custom control architecture meets the following design requirements.

1. The .NET client component can load not only standard controls provided by the HMI, but also the .NET custom controls like user-design controls or controls developed by a third party.

2. The business logic for controls and UI rendering for controls are separated. The business logic resides in the native runtime client component, while the UI rendering is in the .NET client component which is in a separate process. The business logic of controls is designed generically to handle different control events no matter what kind of custom controls are loaded during runtime.

3. All the features used in the architecture are supported in both the .NET Framework 3.5 and the.NET CF 3.5. The prototype is verified on an embedded Panel device with the WinCE 6.0 operating system environment and a standard Windows PC with the Win32 operating system environment.

4. The functionality and efficiency of the architecture are tested. Test results shows that the functionality is complete and operating. The time overhead of the architecture is low enough to prove the architecture has a good efficiency.

The final architecture for the cross-platform .NET custom control is defined as figure 6.1.
The architecture contains a native runtime client component, an out of process COM server component, a .NET client component and an in-process COM server DLL wrapper embedded in the .NET client component.

The native runtime client component is the HMI runtime application. The generic business logic of controls is implemented in the native runtime client component.

The .NET client component holds .NET custom controls. It loads .NET custom controls during runtime. The UI rendering work is done in the .NET client component.

The out of process COM server component provides communication interfaces for both the native runtime client component and the .NET client component. It is declared as a singleton object so that the communication channel can be established.

The in-process COM server wrapper is used to eliminate the marshaling work between the .NET client component and the out of process COM server component, and it is embedded in the .NET client component.

A prototype is implemented to verify the architecture. Verification of the architecture prototype mainly includes two parts: functionalities verification and prototype efficiency, both of which are verified successfully. The results of functionalities verification prove that functionalities of the architecture are complete and the architecture is a solution for
the cross-platform .NET custom control architecture. The results of prototype efficiency demonstrate the architecture has a reasonable efficiency.

### 6.2 Future Work

Currently only .NET custom controls are supported in the architecture. We hope that native custom controls can be supported in future work, which means both .NET custom controls and native custom controls can be loaded in the container in the .NET client component. Because the container is .NET code itself, a proper wrapper between native custom controls and the container should be developed. According to the research of interoperability between managed code and native code in this thesis, it might be possible to find such a solution.

From the time overhead results we can see that most time overhead is caused by the cross-process communication. So it is meaningful to consider minimizing the number of components in separate processes in current architecture. Now the number of components in separate processes is three. We can try to minimize the number of components to two by embedding the out of COM server component to the native runtime client component and verify the functionality and the efficiency of the new architecture. If the new architecture with two components can improve the efficiency, it proves this is a correct way to improve the architecture. Then we can continue trying to minimize the number of components to one by embedding the .NET client component to the native runtime client component by using proper wrapper. Finally, an architecture which can improve the efficiency with complete functionality should be developed.
References


