Programmer's Guide

M5T Framework SAFE – Version 2.1

Proprietary & Confidential

This page is left intentionally blank.

Publication History

Release	Date	Description
01	2007/02/12	Initial Release.
02	2007/03/12	Config section reviewed by M5T, Startup section added to document.
03	2007/04/02	ECom section added.
04	2007/04/17	CServicingThread section added.
05	2007/05/22	Document review
06	2008/04/24	CServicingThread precisions and Resolver folder added.
07	2008/05/12	Asynchronous Socket Factory documentation.
08	2009/09/15	Smart pointers section added.

List of Acronyms

API	Application Program Interface
CAP	Containers, Algorithms and Patterns
CLSID	Class ID
COM	Component Object Model
DNS	Domain Name System
ECOM	Embedded Component Object Model
IETF	Internet Engineering Task Force
IP	Internet Protocol
M5T	M5T (brand name)
PKI	Public Key Infrastructure
PWP	Partners Web Portal
RFC	Request for Comments
RTCP	Real-Time Control Protocol
RTP	Real-Time Protocol
SAFE	Secure and Advanced Feature Edition
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SRTP	Secure Real-Time Protocol
TCP	Transport Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
XML	Extensible Markup Language

Terms and Definitions

Key words The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Status

Work in Progress: An incomplete document, designed to guide discussion and generate feedback, which can include several alternative requirements for consideration.
Draft: A document in specification format considered largely complete, but lacking review. Drafts are subject to substantial change during the review process.
Interim: A document that has undergone rigorous review.
Released: A stable document, reviewed, tested and validated.

List of Figures, Tables, and Code Samples

List of Figures

List of Tables

List of Code Samples

About this Document

Welcome to the M5T Framework SAFE v2.1 Programmer's Guide. This document contains seven sections:

Table 1 - Documentation Structure

Section	Description
Section 1 - Introduction to the M5T Framework SAFE package	Provides a high-level definition of the M5T Framework SAFE product, what it supports and what it requires.
Section 2 - The M5T Framework High-Level Architecture	Overview of each package that makes up the Framework.
Section 3 - Configuration	Explains how the configuration file should be used for proper configuration of the Framework.
Section 4 - Framework Initialization / Finalization	Explains how to perform proper initialization and finalization of the Framework.
Section 5 - ECOM	Introduction to **COM** and **ECOM**. Shows how to work with **ECOM** classes.
Section 6 - CServicingThread and CEventDriven Classes	Introduction to **CServicingThread** and the services that it offers, such as messaging, timers, and socket event monitoring. This section also covers the **CEventDriven** class, which is a helper class used to facilitate the use of the **CServicingThread**.
Section 7 - Asynchronous Socket Factory	Presents the Asynchronous Socket Factory's features.
Section 8 - Smart pointers	Presents the smart pointer classes.
Section 9 - References	Identifies important references for the M5T Framework SAFE.

Table 2 - Text and Symbol Conventions

Convention	Description
`Courier New`	• Sample code • Manager • Interface
Bold	• Request, Response • Field • Package • Header • Method
Italics	• Scenario • Special information • Quotes from a source which is external to the document
ATTENTION!	Indicates important information about the current topic.
CAUTION!	Indicates a potentially hazardous situation which, if not avoided, may result in damage to the equipment or loss of data.

Table of Contents

1. Introduction to the M5T Framework SAFE Package

1.1 What is the M5T Framework SAFE?

The M5T Framework is a suite of tools, algorithms, and patterns that serves as the foundation for building advanced products. The M5T Framework abstracts the operating system and the network access, as well as simplifies threading for all M5T products. It can also be directly accessed and used as a foundation for portable, multi-threaded applications.

The following diagram illustrates the M5T core technology. It shows that all M5T components are built over the M5T framework.

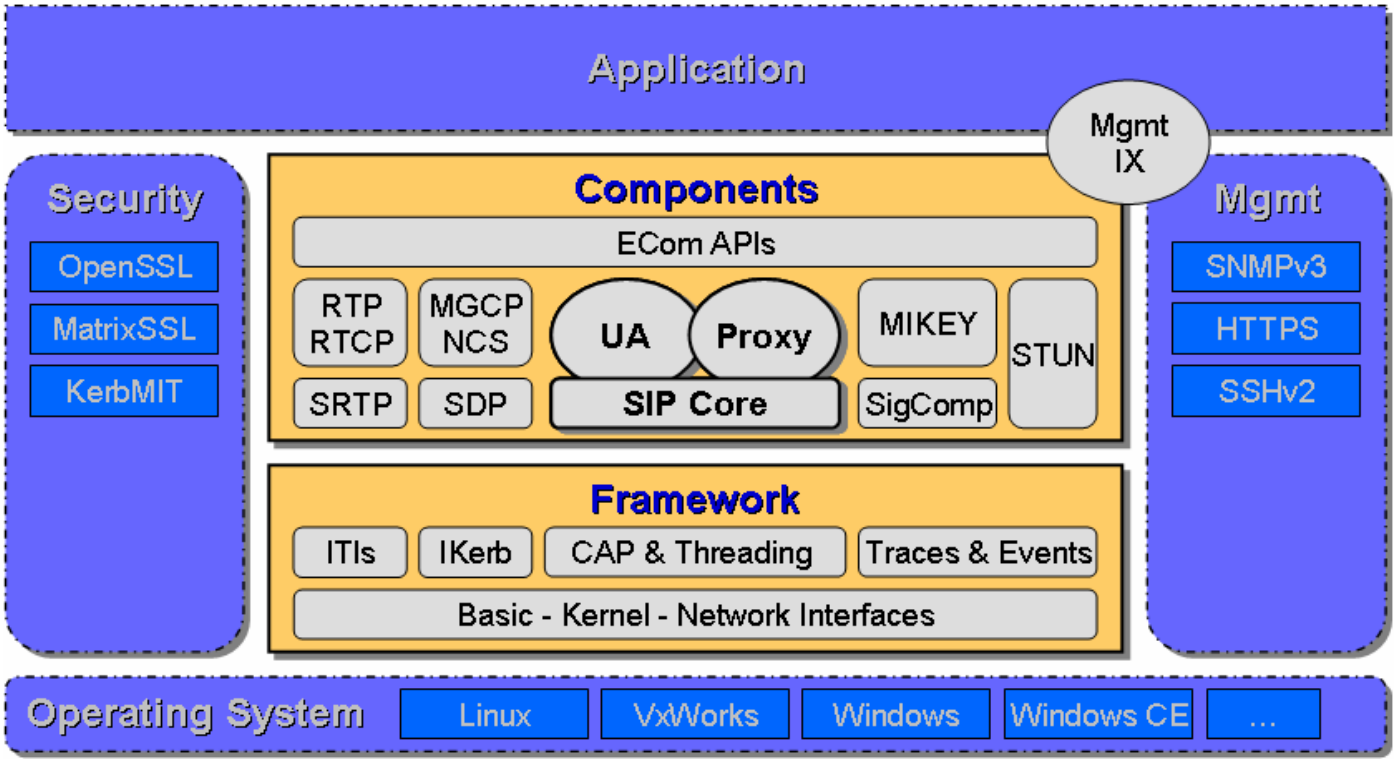


Figure 1: M5T Core Technology

The following are the objectives aimed by the framework package:

- Knowledge base.
- Source code reuse throughout various projects.
- Interface unification and flexibility.
- Abstraction.
- Portability and scalability.
- Quality.
- Accelerate application development.

The classes and public methods of the framework are extensively described in a document named “M5T Framework SAFE v2.1 - API Reference.pdf”. The purpose of this programmer’s guide is different: it is meant to explain to a programmer unfamiliar with M5T packages the main concepts and mechanisms behind the Framework package. Code examples are also provided to allow a better comprehension of how the Framework package should be used.

After you read this document, if you still have technical questions, you should refer to the API reference document or the source code itself. If further questions remain, you could contact M5T via the M5T Partners Web Portal (http://www.m5t.com/support-center.php).

1.2 Supported Features

From a high-level point of view, the list of features offered by M5T Framework includes:

- **Operating system abstraction:** thread, semaphore, mutex, timers, etc.
- **Network abstraction:** UDP/TCP sockets, IPv4/IPv6 socket addresses, etc.
- **Templated containers:** list, vector, tree, queue, stack, etc.
- **Design patterns:** singleton, memory allocator, etc.
- **Advanced mechanism:** Servicing thread for centralized asynchronous message, timer events, and socket events handling.
- **Protocol abstraction:** Kerberos, TLS, etc.
- **Cryptography and security abstraction:** AES, MD5, SHA1, SHA1-MAC, RSA, CRC, Hash, Diffie-Hellman, PKCS7, PKCS12, PKI, Certificate, Certificate chain, public and private key, etc.
- **Portable DNS resolver** and caching aligned on RFCs 1035 and 2181, and caches queries described in RFCs 1035, 2782, and 3403.
- **Other mechanisms:** tracing, assertion, etc.

1.3 Requirements

1.3.1 Software

ANSI C++ Compiler that supports placement new operator and multiple inheritance.

1.3.2 OS and Architecture

Validated combinations of OS / Architecture / Compiler:

- Linux 2.4 / ix86 and ppc / gcc 3.1.1.
- Linux 2.6 uclibc / mips / gcc 3.4.4.
- Linux 2.6 / amd64 / gcc 4.1.2.
- WinXP, Win2K3, Win2K and Vista / ix86 / MSVC6, MSVC7 and MSVC8.
- VxWorks 5.4 / mips / gcc 2.9.5.
- VxWorks 5.5, VxWorks 6.5 and VxWorks 6.6 / ix86 (pentium) / gcc 2.9.6.
- Nucleus / nios2 / gcc 3.4.1.
- Symbian 9.1 / arm / gcc 3.4.3.
- None / DSP emulator / cl6x (Code Composer Studio 2.2).
- WinCE5 / arm / MSVC8.

Additional operating systems/architectures can easily be supported if the above requirements are met.

1.4 Dependencies

Depending on the features used, this product version requires:

- M5T MITKerberos v1.0.2 (Kerberos functionality).
- M5T OpenSSL v1.0.4 (Encryption and TLS functionalities).
- M5T Expat v1.1.5 (XML functionality).
- M5T Regex v1.0.1 (Regular Expression functionalities).
- M5T LIB_ SYMBIAN v1.0.1 (Symbian 9.1 builds only).
- M5T LIB_NUCLEUS v1.0.1 (Nucleus builds only).
- M5T LIB_WCECOMPAT v1.0.1.3 (Microsoft Windows CE builds only).
- The Microsoft platform SDK 2003 (February 2003 version) is required for MSVC 6.0 builds.
- The Microsoft platform SDK 2003 or later is required for Microsoft .NET builds.
- The following Nucleus products: Nucleus PLUS, Nucleus Posix, Nucleus NET, Nucleus Ethernet driver, Nucleus C++ BASE.

Mandatory libraries for all Windows platforms are:

- ws2_32.lib: Required for UDP and TCP socket support.
- dnsapi.lib: Required for SRV queries that are supported only on Windows NT, Windows 2000, and higher. They are available from Microsoft's platform SDK.

Optional libraries for all Windows platforms are:

- comerr.lib, crypto.lib, gssapi.lib, krb5.lib, profile.lib, secur32.lib : Required for Kerberos support.
- libeay32.lib, ssleay32.lib : Required for OpenSSL crypto algorithms and TLS support.

Mandatory libraries for all Windows platforms, other than Windows CE:

- winmm.lib : Required for CTimer.
- traffic.lib: Required for QoS.

Mandatory library for Windows CE:

- mmtimer.lib : Required for CTimer.

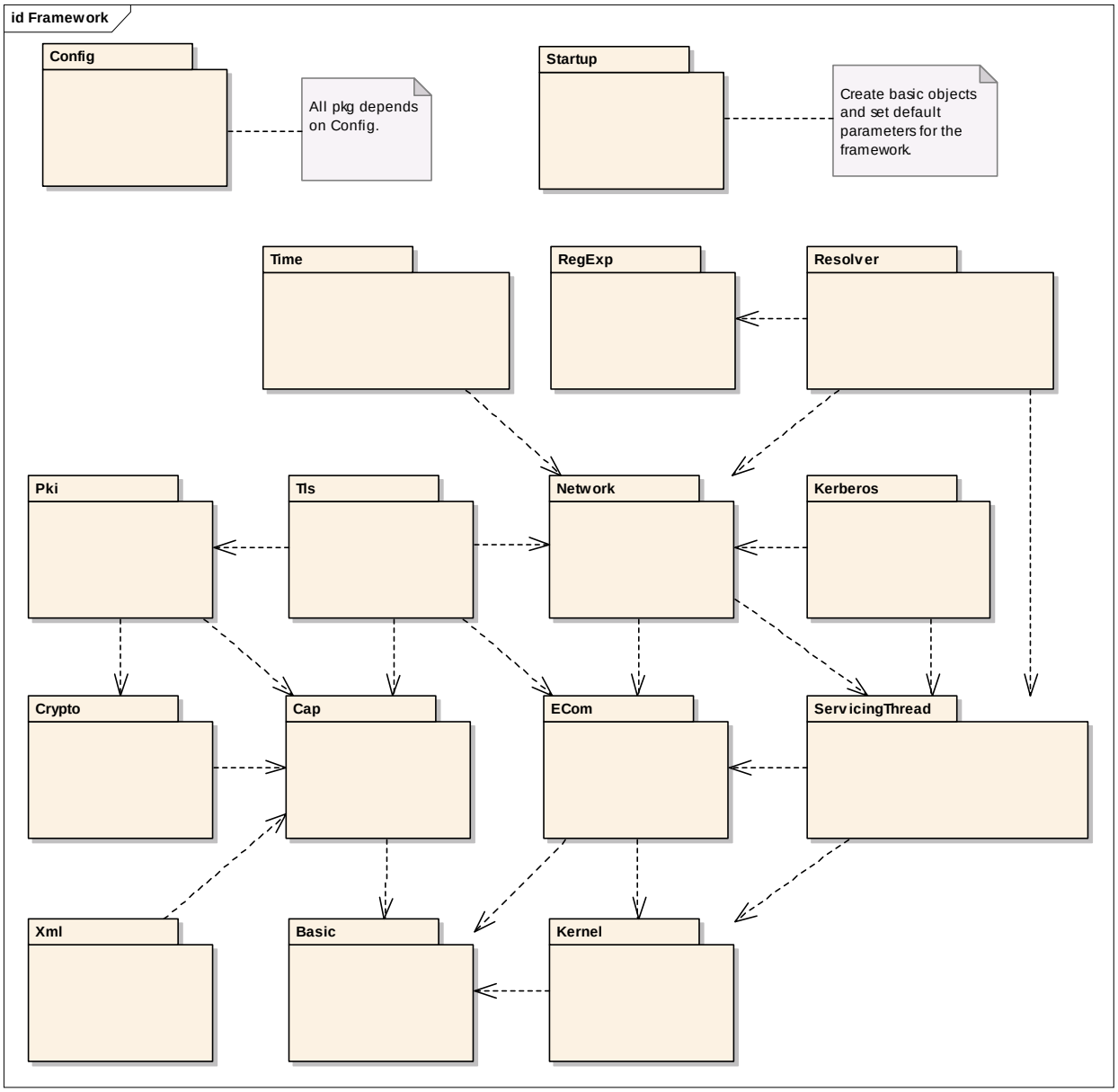
2. The M5T Framework High-Level Architecture

The M5T Framework SAFE product is separated into a set of packages, with each one having its own logical boundaries or sets of responsibilities. Each package has its own directory where the source and header files are located, with the directory name being the same as the package's name.

The following sections provide a brief overview of each of these packages.

2.1 Packages Diagrams

Figure 2: Package Diagrams



2.2 Packages Descriptions

2.2.1 Basic

In this package, you will find:

- Basic type definition.
- Helper macros.
- Compiler validation.
- OS validation.
- Tracing macros.
- Assertion macros.
- mxt_result definition and related macros.
- Portable snprint.
- Smart pointers.

2.2.2 Cap

This package implements containers, algorithms, and patterns used extensively throughout the framework and the other M5T components. The following is a short list of the functionalities offered:

- Basic containers (String, Blob, Marshaler).
- Templated containers (Vector, List, VList, Queue, Stack, Map).
- Patterns (Singleton).

2.2.3 Config

- Contains the functionalities to allow global configuration and framework-specific configuration.
- Framework Basic configuration based on OS / Compiler / Architecture.
- All other packages have a dependency on this package.
- See Section 3: Configuration for details on how to use this package.

2.2.4 Crypto

- Offers various functionalities necessary to implement security features, such as:
  - Symmetric-key encryption (AES).
  - Public-key encryption (RSA).
  - Key sharing protocol (Diffie-Hellman).
  - Hash algorithms (MD5, SHA-1).
  - CRC calculation.
  - Secure pseudo-random generator.

2.2.5 ECom

- The ECOM (Embedded Component Object Model) interfaces are inspired by the Microsoft COM standard with some adaptations for embedded applications.
- See section 5 for details on this package.

2.2.6 Kerberos

Kerberos is a computer network authentication protocol. The M5T implementations of the protocol have the following characteristics:

- Abstracts MIT Kerberos.
- Abstracts GSS API.

2.2.7 Kernel

The purpose of this package is to abstract standard OS services, such as:

- Threading.
- Synchronization objects (semaphores, mutexes).
- Timers.

It also offers memory services. All of these functionalities are supported on a wide variety of operating systems.

2.2.8 Network

- Abstracts Network objects:
  - Synchronous and asynchronous sockets.
  - UDP/TCP protocols.
  - Asynchronous socket factory

2.2.9 Pki

- This package contains Public Key Infrastructure helper and abstraction classes, such as:
  - Certificate.
  - Certificate chain.
  - Private Key.
  - Public Key.

2.2.10 RegExp

This package contains classes that abstract calls to a regular expression library supporting POSIX Extended Regular Expressions.

2.2.11 Resolver

- This package contains the classes responsible for the DNS queries. Specifically, it contains:
  - Synchronous and asynchronous resolvers:
    - A and AAAA
    - CNAME
    - PTR
    - SRV
    - NAPTR
    - ENUM
    - Recursion when the DNS server does not support it
  - Resolver cache
  - Portable resolver
  - Various resolvers based on the operating system resolver

2.2.12 ServicingThread

This package offers high-level abstraction of:

- Threading.
- Messaging.
- Timer mechanism.
- Socket management.

2.2.13 Startup

This package contains the classes responsible to initialize and finalize the Framework. For more details, refer to section 4: Framework Initialization / Finalization.

2.2.14 Time

Offers various time-based services:

- Current date and time queries.
- SNTP.
- Time zones.

2.2.15 TLS

This package offers an abstraction of a TLS socket (client or server). It uses the functionalities of the OpenSSL library.

2.2.16 XML

Offers XML parsing services and abstracts EXPAT.

3. Configuration

There is a default configuration for each M5T package, including the M5T Framework. The provided default configuration can be tailored by a software developer to match the specific needs of an application. Configurations settings range from choosing the application's build target, tracing levels to even specify the size of internal data structures.

M5T provides a flexible configuration mechanism that permits applications to configure M5T software modules at different levels, while minimizing the number of files being rebuilt after changing a configuration option. This section describes how to use that configuration mechanism.

3.1 Configuration

The M5T Framework has a **Config** directory, which contains the configuration specific to the module. Configuration options are pulled from the files found in that directory.

There are two main configuration categories: the **global configuration** and the **framework-specific** configuration settings. Both of these sections have pre-configuration and post-configuration levels.

3.1.1 Global Configuration

Global Configuration is done via the **PreMxConfig.h**, **MxConfig.h**, and **PostMxConfig.h** files. These files pertain to the platform, operating system, and compiler settings. They also set any global defines required by the application.

3.1.1.1 MxConfig.h

The **MxConfig.h** file is different from other configuration files since it does not configure a single module, but instead, defines configuration options that are used throughout all of the M5T software components. All of M5T header files include **MxConfig.h** before any other files.

To know what configuration settings can be used, this file should be used as a reference. To enable a specific configuration setting, the **PreMxConfig.h** / **PostMxConfig.h** files should be used.

3.1.1.2 PreMxConfig.h

The main configuration file (**MxConfig.h**) uses the pre-configuration file (**PreMxConfig.h**) to pull application-specific configuration options and apply them. The following is a partial list of options that could be set in this file:

- Select the OS and compiler to use.
- Set the architecture of the host (architecture family, endianness, etc.).
- Configure the tracing mechanism.
- Memory allocation configuration.

All configuration settings are well documented in the **MxConfig.h** file and the API reference document.

For example, the following defines could be set in the **PreMxConfig.h** file.

```
#define MXD_ENABLE_NAMESPACE
#define MXD_ASSERT_ENABLE_ALL
#define MXD_TRACE_ENABLE_ALL
#define MXD_RESULT_ENABLE_ERROR_MESSAGES
#define MXD_RESULT_ENABLE_SHARED_ERROR_MESSAGES
#define MXD_RESULT_ENABLE_MITOSFW_ERROR_MESSAGES
#define MXD_MEMORY_ALLOCATOR_ENABLE_SUPPORT
#define MXD_MEMORY_ALLOCATOR_STATISTICS_ENABLE_SUPPORT
#define MXD_MEMORY_ALLOCATOR_PROTECTION_ENABLE_SUPPORT
#define MXD_MEMORY_ALLOCATOR_BOUND_CHECK_ENABLE_SUPPORT
#define MXD_MEMORY_ALLOCATOR_EXTRA_INFORMATION_ENABLE_SUPPORT
#define MXD_MEMORY_ALLOCATOR_MEMORY_TRACKING_ENABLE_SUPPORT
```

Code Sample 1 - Example of a PreMxConfig Define Section

ATTENTION!
Due to the dependency of all M5T files on **MxConfig.h**, any modifications done in the pre-configuration file require to re-compile all of the M5T files.

3.1.1.3 PostMxConfig.h

The **PostMxConfig.h** file allows settings specified in the **MxConfig.h** file to be overridden without having to edit **MxConfig.h**. This can be useful for an application that uses multiple settings across different platforms. The inclusion of the **PostMxConfig.h** is enabled by the macro **MXD_POST_CONFIG** in the **PreMxConfig.h** file:

```
#define MXD_POST_CONFIG
```

Code Sample 2 - Post-Configuration Macro

The **PostMxConfig.h** file can be located within the **"../Sources/Config"** folder, but is only required to be placed somewhere in the compiler search path.

3.1.2 Framework-Specific Configuration

Framework-specific configuration is done via the **FrameworkCfg.h**, **PreFrameworkCfg.h**, and **PostFrameworkCfg.h** files.

3.1.2.1 FrameworkCfg.h

This file is the main configuration file for the M5T Framework and it contains the default configuration settings. It is important to note that this file is not meant to be modified, but only to act as a guide for the pre- and post-configuration files. The **FrameworkCfg.h** file contains documentation on settings that can be applied via the **PreFrameworkCfg.h** file. It is helpful to look within the **FrameworkCfg.h** file to see which setting can be used and what they are used for. The **FrameworkCfg.h** file is located at **"../Sources/Config"** within the Framework module.

3.1.2.2 PreFrameworkCfg.h

To override the default framework configuration settings, you must use the **PreFrameworkCfg.h** file. Please note that **PreFrameworkCfg.h** is not packaged with M5T's distributions and must always be created. This file should be placed within the "**../Source/Config"** folder along with the **FrameworkCfg.h** file. This file must be provided, even if the application is only using the default settings found in the **FrameworkCfg.h** file. This is meant to force an acknowledgement that the application has been configured as desired.

All possible settings are fully described in **FrameworkCfg.h** and also in the API reference document. These settings need to be enabled in **PreFrameworkCfg.h**. This file is included by **FrameworkCfg.h** to retrieve any framework-specific configurations prior to applying the default settings.

Having the appropriate set of defines not only gives the application the required functionality, but also allows to reduce the application code size. For example, if the framework containers (**CString**, **CVector**, etc.) are not needed, the **MXD_CAP_ENABLE_SUPPORT** macro shouldn't be defined to prevent containers from being compiled.

This is a typical set of defines that could be found in **PreFrameworkCfg.h**:

```
// Crypto configuration: use the framework algorithms
#define MXD CRYPTO ALL MITOSFW

// Use M5T containers
#define MXD CAP ENABLE SUPPORT

// Use M5T ECOMs
#define MXD ECOM ENABLE SUPPORT

// Enable Servicing Thread module
#define MXD SERVICING THREAD ENABLE SUPPORT

// Enable Time module
#define MXD_TIME_ENABLE_SUPPORT
```

Code Sample 3 - Example of a PreFrameworkCfg.h define Section

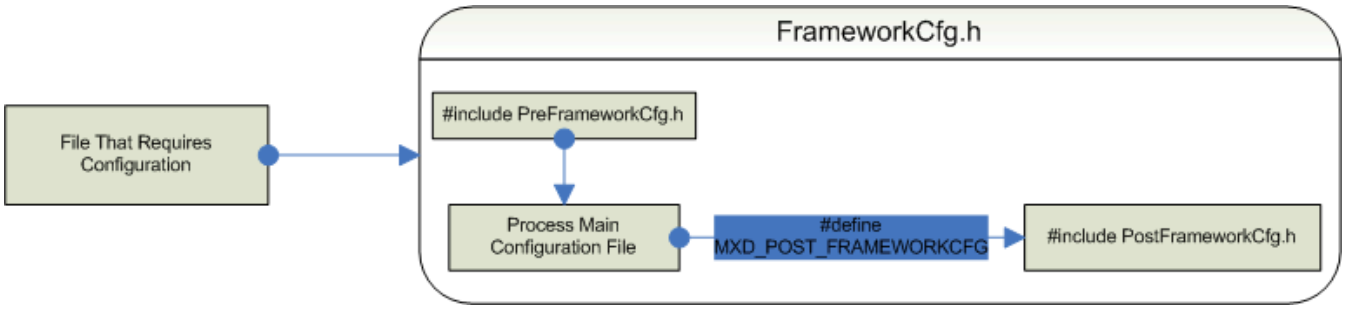
3.1.2.3 PostFrameworkCfg.h

The post-configuration file is also an application-specific configuration file that is created to allow the application to further configure the package based on the settings that were actually applied by the pre-configuration and main-configuration files.

Unlike the pre-configuration file, the post-configuration file is not automatically included by the main-configuration file. The application must make sure to define the proper macro in order to force the inclusion of the post-configuration file. Therefore, this file needs to be created if some post-configuration is required. The inclusion of the **PostFrameworkCfg.h** is enabled by the macro **MXD_POST_FRAMEWORKCFG** in the **PreFrameworkCfg.h** file.

Figure 3 - Framework Configuration Relationship

The **PostFrameworkCfg.h** file can be located within the **"../Sources/Config"** folder, but is only required to be placed somewhere in the compiler search path to permit the retrieval of the application-specific configuration options.



4. Framework Initialization / Finalization

It is important for M5T software packages to have precise control of the order in which internal data structures are initialized and finalized. By doing so, the following issues are eliminated:

1. If code is used in a DLL and released using the **FreeLibrary** function, it will cause memory leaks since all global and static variables are not being de-allocated.

2. Dependencies between global variables can cause bugs or crashes, because of an incompatible linking order.

Each M5T software package, including the framework, implements initializers. They are located in a folder named **Startup**. This folder contains at least two files; **C[PACKAGE_NAME]Initializer.h** and **C[PACKAGE_NAME]Initializer.cpp**. Therefore, the Framework has the **CFrameworkInitilizator.h** and **CFrameworkInitilizator.cpp** file in its **Startup** package..

The initializer classes are responsible to initialize and finalize the software package and its dependencies. For example, the M5T **SipStack** has a dependency on the Framework, therefore **CSipStackInitializer::Initialize** calls **CFrameworkInitializer::Initialize**. In the same fashion, **CSipStackInitializer::Finalize** calls **CFrameworkInitializer::Finalize**.

The following section describes how to initialize and finalize the Framework.

4.1 CFrameworkInitializer

Before any functionality is used in the Framework, the Framework needs to be initialized. The **FrameworkInitializer::Initialize** method initializes core components such as servicing threads, sockets containers, tracing mechanisms, and data marshalers.

The **CFrameworkInitializer::Finalize** properly cleans the Framework core components and needs to be called when the application exits. Failing to do so might leave some objects in an improper state and even cause memory and resource leaks on some platforms. In addition, calling **Finalize** also allows the application developer to gather various statistics about the Framework usage. This information is obtained via the **SFrameworkFinalizeInfo** structure.

The **Initialize** and **Finalize** methods may be called multiple times, therefore the **CFrameworkInitializer** class must use a reference counting mechanism to not initialize more than once and to not finalize until the last user of the framework has called **Finalize**. For instance, when the M5T **RTP** and the M5T **SIP** packages are being used, the **CFrameworkInitializer::Initialize** method is called twice, but the core components is initialized only once on the first call. Resources are then unallocated once the second call to the **Finalize** method is made.

The **CFrameworkInitializer** class is not thread safe. If more than one call to the **Initialize** or **Finalize** methods are required, these calls must come from either the same thread or be protected against concurrent access from multiple threads. Note that the framework's own thread synchronization objects (the **CMutex** or the **CSemaphore**) must not be used at that time since the framework has not been initialized.

If multiple M5T packages are being used in different threads, for example when the M5T **SIP** and **RTP** packages are initialized, calls to **CSipStackInitializator::Initialize** and **CRtpInitializator::Initialize** must be made thread safe since they end up calling **CFrameworkInitializer::Initialize**.

4.2 Tracking Memory Usage Using SFrameworkFinalizeInfo

By using the **CMemoryAllocator::SMemoryStatistics** contained in the **SFrameworkFinalizeInfo** structure (i.e.: **SFrameworkFinalizeInfo::m_stMemoryStatistics** member variable), it is possible to know exactly how much memory has been allocated by the framework. The number of calls to the new and delete operators, the biggest allocated memory block, and more importantly, the peak memory usage can also be known. This data can help establish the memory requirements for a specific application.

The information contained in the **SFrameworkFinalizeInfo** structure can help a developer track down memory leaks pertaining to the framework. This requires that the **MXD_MEMORY_ALLOCATOR_MEMORY_TRACKING_ENABLE_SUPPORT** and **MXD_MEMORY_ALLOCATOR_EXTRA_INFORMATION_ENABLE_SUPPORT** macros have been enabled before the framework initialization. These macros need to be defined in the **PreMxConfig.h** file.

Memory leaks can be suspected when the number of calls to the new operator is greater than the number of calls to the delete operator. The following code example illustrates how to obtain memory statistics and look for memory leaks. It displays the Framework's peak memory usage. For an unreferenced memory block, it also shows the filename and line number in which the memory allocation was made.

```
int main()
{
    // Initialize the Framework.
    CFrameworkInitializer::Initialize();

    // Activate traces for all packages
    MxTraceEnableNode("/", true);

    MX_TRACE0(0,
              g_stFramework,
              "Start test execution");

    // Force a memory leak to show how the memory statistics works.
    CString* pstrName = MX_NEW(CString)("This is a test");

    // Finalize the Framework.
    CFrameworkInitializer::SFrameworkFinalizeInfo stFinalizeInfo;
    CFrameworkInitializer::Finalize(&stFinalizeInfo);

#if defined(MXD_MEMORY_ALLOCATOR_STATISTICS_ENABLE_SUPPORT)

    // Note: we must use printf since the tracing mechanism of the framework is
    // no longer available

    // A structure holding memory allocation statistics.
    CMemoryAllocator::SMemoryStatistics stMemoryStatistics =
          stFinalizeInfo.m_stMemoryStatistics;

    // Check for memory leaks
    if (stMemoryStatistics.m_uNewCallCount != stMemoryStatistics.m_uDeleteCallCount)
    {
        printf("Memory leak detected! \n");
    }

    printf("Peak heap memory usage: %u \n",
           stMemoryStatistics.m_uPeakMemoryUsage);
#endif

#if defined(MXD_MEMORY_ALLOCATOR_MEMORY_TRACKING_ENABLE_SUPPORT)

    unsigned int uNbOfMemoryBlocks = stFinalizeInfo.m_uNumberOfAllocatedMemoryBlocks;
```

```
    printf("Number of memory blocks still allocated: %u \n",
           uNbOfMemoryBlocks);

    unsigned int i = 0;
    CFrameworkInitializer::SFrameworkFinalizeInfo::SMemoryInfo stMemBlockInfo;

    // printout information on all memory blocks that are still allocated
    for ( ; i < uNbOfMemoryBlocks; i++)
    {
        // Get next leaked memory block
        stMemBlockInfo = stFinalizeInfo.m astMemoryBlockInfo[i];

        printf("Memory block # %u | Type: %s, File: %s Line: %u\n",
               i,
               stMemBlockInfo.m pszType,      // Type of data contained in the memory block.
               stMemBlockInfo.m pszFilename,  // File in which the memory allocation was
                                              // done.
               stMemBlockInfo.m uLineNumber); // Line number in the file where allocation
                                              // was made.
    }
#endif
    return 0;
}
```

Code Sample 4 - Example of How to Use the SFrameworkFinalizeInfo Structure

You will find below the output generated when the preceding code sample is executed. The information about each memory blocks that were not released should allow a developer to quickly correct the memory leak detected:

```
   <12> |||0|239734540|1|Start test execution
   <12> |||0|239734540|2|Test completed!
   <14> |||0|239734540|3|EComFactory::UnregisterECom(0x8292fec)
   <15> |||0|239734540|4|EComFactory::UnregisterEComExit(0)
   <14> |||0|239734540|5|CCrypto(0x82eb120)::~CCrypto()
   <15> |||0|239734540|6|CCrypto(0x82eb120)::~CCryptoExit()

   Memory leak detected!
   Peak heap memory usage: 86320
   Number of memory blocks still allocated: 2
   Memory block # 0 | Type: uint8 t, File: ../../Sources/Cap/CString.cpp Line: 738
   Memory block # 1 | Type: CString, File: ../../TestCases/TestFw21.cpp Line: 321
```

Code Sample 5 - Output Generated When Executing Code Example

If you need additional information about the **SFrameworkFinalizeInfo**, **SMemoryStatistics**, or **SMemoryInfo**, please refer to the Startup section of the Framework API Reference.

5. ECOM

ECOM stands for Embedded Component Object Model. It is based on Microsoft's Component Object Model. For this reason, it is recommended that any programmer working with **ECOM** has previous experience or knowledge of **COM**. For more information on **ECOM**, please refer to the Framework API Reference document (**ECOM** Section).

The difference between **ECOM** and **COM** is that **ECOM** is a more lightweight design than Microsoft's **COM** and only allows static linking (i.e. no dynamic linking). **ECOM** is also in-process only, meaning that the **ECOM** object is loaded in the same process as the code that is calling it. For more comparisons between **ECOM** and **COM**, please refer to the Framework API (**ECOM** Section) and any documentation pertaining to Microsoft's **COM**.

As previously seen in *Figure 1: M5T Core Technology*, the **ECOM** API sits on top of all the other components (i.e. **RTP**, **STUN**, **UA**, etc.). The reason is because these M5T packages all have at least one class which is **ECOM**.

5.1 Advantages of Using ECOM

There are several advantages to using **COM** or **ECOM**:

- Dynamic discovery of an object's supported interfaces.
- Completely hides an object's implementation details.
- Component reusability in many environments.
- Reference counting (see Section 5.3 for more details).

To use any of the **ECOM** functionalities, you need to define **MXD_ECOM_ENABLE_SUPPORT** in your **PreFrameworkCfg.h** configuration file.

5.2 Presentation of the Main ECOM Methods

5.2.1 CreateEComInstance

To create an **ECOM** class instance, a developer simply needs to use the **CreateEComInstance** method. This method creates an instance of a **ECOM** class in memory. There is an interface pointer returned through a **void**** parameter that is the first reference to this **ECOM** interface. Therefore, at this point, the **ECOM** object automatically has a reference count of one.

Each **ECOM** class must have its own class identifier, called **CLSID**, which is defined in the **EComCLSID.h** file present in the same package as the class. The **CLSID** is made up of the **ECOM** class name and a **CLSID_** prefix. For example, the Servicing Thread **ECOM** class identifier is **CLSID_CServicingThread**. Moreover the **CLSID** of an **ECOM** object is the unique public symbol referring to the implementation details of a component enforcing abstraction and limiting use of the component to its supported interfaces.

Calling **CreateEComInstance** requires at least two parameters: the class identifier (**CLSID**) of the **ECOM** class to create and an interface pointer used to store the requested interface pointer. The second argument, **pOuterIEComUnknown**, could be set to **NULL**, except when you need to create an **ECOM** class that is aggregated. More information on **ECOM** aggregation is available in section 5.4. The **CreateEComInstance** method returns **resS_OK** if the instance is created successfully and an error otherwise. For the list of possible error codes, please refer to the method header or the Framework API.

NOTE
On success, **CreateEComInstance** calls **AddIfRef** internally. Read more about reference counting in Section 5.3.

5.2.2 QueryIf

To query the interface of an **ECOM** instance, the **QueryIf** method must be used. This method is used to query an object for a supported interface. The method requires a single parameter, which is the address of a pointer to the requested interface type, in order to store the result. Similarly to **CreateEComInstance**, the **QueryIf** method returns a **mxt_result** to indicate the success of the request.

NOTE
On success, **QueryIf** automatically calls **AddIfRef** to increase the reference count on the object. Read more about reference counting in Section 5.3.

There are several rules that need to be remembered when using **QueryIf**:

1. You always get the same **IEComUnknown**.
2. You can always get an interface if you got it before.
3. You can always query for the interface you already have.
4. You can always get back to where you started.
5. If you can query an interface from another interface, then you can query this interface from all other interfaces supported by this object.

5.2.3 AddIfRef / ReleaseIfRef

The **AddIfRef** method is used to increment the reference count on the **ECOM** object. The **AddIfRef** returns the increased reference count value. However, this value is only meant for providing debug information.

On the other hand, the **ReleaseIfRef** method is used to decrement the reference count on the **ECOM** object. The **ReleaseIfRef** returns the decreased reference count, which is only meant to provide debug information.

The **AddIfRef** and **ReleaseIfRef** methods enable control on an ECOM object's reference count as the mechanism to manage the object's lifecycle. Hence, when the reference count of an object reaches zero, the object deletes itself from memory. For more information about when the **AddIfRef** and **ReleaseIfRef** methods should be used, refer to the reference counting rules in Section 5.3

Below is an example of how the various **ECOM** methods described should be used.

```
// Create a Servicing Thread instance and get it's IActivationService
// interface pointer (ref count = 1).
IActivationService* pActivationService = NULL;
mxt result res = CreateEComInstance(CLSID CServicingThread, NULL, &pActivationService);

if (MX RIS S(res))
{
    // Activate the CServicingThread. A new thread will be created.
    res = pActivationService->Activate();

    // Query it's ISocketService interface ()
    ISocketService* pSocketService = NULL;
    res = pActivationService->QueryIf(&pSocketService);
    // ref count = 2

    if (MX RIS S(res))
    {
        // Copy an interface pointer
        ISocketService* pSocketService2 = pSocketService;
        pSocketService->AddIfRef();
        // ref count = 3.
```

```
            // ...

            // Pointer no longer necessary, release it.
            pSocketService2->ReleaseIfRef();
            pSocketService2 = NULL;
            // ref count = 2.
        }

        // We no longer need the ISocketService pointer, release it.
        if (pSocketService != NULL)
        {
            pSocketService->ReleaseIfRef();
            pSocketService = NULL;
            // Ref count = 1.
        }
    }

    // Release the CServicingThread.
    if (pActivationService)
    {
        uRefCountValue = pActivationService->ReleaseIfRef();
        pActivationService = NULL;

        if (uRefCountValue != 0)
        {
            printf("ECOM object not released.  Ref count = %u \n", uRefCountValue);
        }

        // Ref count is now set to zero, ECOM instance is released.
    }
```

Code Sample 6 - CreateEComInstance, QueryIf, AddIfRef, and ReleaseIfRef Example (ECOM)

5.3 Reference Counting Rules

Proper reference counting is crucial since it is the mechanism used to control the lifetime of each **ECOM** object. Each **ECOM** object maintains its own reference count corresponding to the number of interface pointer references currently held and being used on one particular **ECOM**. When the reference count of an **ECOM** object reaches zero, the **ECOM** object frees itself from memory by calling **MX_DELETE(this).**

Keeping the reference count right is important to avoid memory leaks. Here are a few rules to follow to achieve adequate reference counting:

- A call to **ReleaseIfRef** must be made on an ECOM interface pointer before making it invalid when the interface pointer is no longer needed. In the end, there must be a call to **ReleaseIfRef** matching each successful call to **CreateEComInstance**, **QueryIf**, and **AddIfRef**.
- A call to **AddIfRef** must be made on an ECOM interface pointer each time a copy of the pointer must be kept.
- A call to **AddIfRef** must be made on an ECOM interface pointer that is returned by a method as an OUT parameter. The reason behind this rule is to make sure the caller will get a valid interface pointer on return from the method by avoiding a possible transition of the reference count to 0 before the caller gets the time to call **AddIfRef**. The methods **CreateEComInstance** and **QueryIf** are good examples of such a behaviour.

5.4 ECom Aggregation and Containment

It is an error for an ECOM to inherit from another ECOM. In cases when functionality provided by an ECOM is needed by another ECOM, containment and aggregation can be used. Containment and aggregation are techniques in which an outer component uses an inner component.

ECOM containment is very simple. In the case of containment, the outer component is a client of the inner component. The outer component contains pointers to interfaces on the inner components and it implements its own interfaces by using the

interfaces of the inner component. The outer component can also re-implement an interface supported by the inner component by forwarding calls to the inner component.

In the case of ECOM aggregation, the outer component does not re-implement the inner components interface but passes the inner component's interface pointer directly to the client. The client calls the interface belonging to the inner component directly, but it does not know it is talking to two different components.

For example, let's assume the following two interfaces:

```
 class IExampleOne : public IEComUnknown
{
// Published Interface
//-------------------------
public:

    MX_DECLARE_ECOM_GETIID(IExampleOne);

    virtual void IExampleOneMethod() = 0;

// Hidden Methods
//-------------------------
protected:

    // << Constructors / Destructors >>
    //-----------------------------------------
    IExampleOne(){}
    virtual ~IExampleOne(){}

private:

    // Deactivated Constructors / Destructors / Operators
    //-----------------------------------------------------
    IExampleOne(const IExampleOne& from);
    IExampleOne& operator=(const IExampleOne& from);
};

class IExampleTwo : public IEComUnknown
{
// Published Interface
//-------------------------
public:
    MX DECLARE ECOM GETIID(IExampleTwo);

    virtual void IExampleTwoMethod() = 0;

// Hidden Methods
//-------------------------
protected:

    // << Constructors / Destructors >>
    //-----------------------------------------
    IExampleTwo(){}
    virtual ~IExampleTwo(){}

private:

    // Deactivated Constructors / Destructors / Operators
    //-----------------------------------------------------
    IExampleTwo(const IExampleTwo& from);
    IExampleTwo& operator=(const IExampleTwo& from);
};
```

Code Sample 7 – ECOM Example Interfaces

Assume **IExampleOne** is implemented by an ECOM class, **CExampleOne**. Another ECOM class, **CExampleTwo**, is needed that implements **IExampleTwo** but also needs functionality provided by **CExampleOne**. This can be done using containment or aggregation.

Here is an example on how to use containment to implement CExampleTwo.

```
class CExampleTwo :    public CECom DelegatingUnknown,
                       public IExampleOne,
                       public IExampleTwo
{
// Published Interface
//-------------------------
public:
    // << Constructors / Destructors >>
    //-----------------------------------------

    // The constructor will always have one parameter as shown below
    CExampleTwo(IECom Unknown* pOuterIECom Unknown = NULL);
    virtual ~CExampleTwo();

    // << Stereotype >>
    //-----------------------------------------
    static mxt result CreateInstance( IN IECom Unknown* pOuterIECom Unknown,
                                      OUT CECom Unknown** ppCECom Unknown);

    // The following statement is essential, it provides the default and unique
    // implementation of the IECom Unknown interface from which every other interface
    // inherits.
    MX DECLARE DELEGATING IECOMUNKNOWN

    //-----------------------------------------
    //  Override CECom DelegatingUnknown
    //-----------------------------------------
    virtual mxt result InitializeInstance();
protected:
    virtual void UninitializeInstance(OUT bool* pbDeleteThis);

public:
    virtual mxt result NonDelegatingQueryIf(IN mxt iid iidRequested, OUT void**
          ppInterface);

    //-----------------------------------------
    //  Override IExampleOne interface methods
    //-----------------------------------------
    virtual void IExampleOneMethod();

    //-----------------------------------------
    //  Override IExampleTwo interface method(s)
    //-----------------------------------------
    virtual void IExampleTwoMethod();

// Hidden Methods
//-------------------------
protected:
private:

    // Deactivated Constructors / Destructors / Operators
    //-------------------------------------------------
    CExampleTwo(const CExampleTwo& from);
    CExampleTwo& operator=(const CExampleTwo& from);

// Hidden Data Members
//-------------------------
protected:
private:
```

```
    // We must keep a pointer to the object we are containing
    IEComUnknown* m pInnerIEComUnknown;
};

mxt result CExampleTwo::CreateInstance(IN IEComUnknown* pOuterIEComUnknown,
                                       OUT CEComUnknown** ppCEComUnknown)
{
    *ppCEComUnknown = MX NEW(CExampleTwo)(pOuterIEComUnknown);
    if ( *ppCEComUnknown == NULL )
    {
        return resFE OUT OF MEMORY;
    }
    return resS OK;
}

//============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ==============================================
//============================================================================
CExampleTwo::CExampleTwo(IEComUnknown* pOuterIEComUnknown):
    CEComDelegatingUnknown(pOuterIEComUnknown),
    m pInnerIEComUnknown(NULL)
{
}

CExampleTwo::~CExampleTwo()
{
}


//============================================================================
//====  CEComUnknown  ========================================================
//============================================================================
mxt result CExampleTwo::InitializeInstance()
{
    // Create inner component
    mxt result result = CreateEComInstance( CLSID CExampleOne,
                                            NULL,
                                            &m pInnerIEComUnknown);

    // Validate the creation
    MX ASSERT( MX RIS S(result) && m pInnerIEComUnknown != NULL);
    return result;
}

void CExampleTwo::UninitializeInstance( OUT bool* pbDeleteThis )
{
    CEComUnknown::UninitializeInstance( pbDeleteThis );

    // Release our inner object so it is deleted from memory
    m pInnerIEComUnknown->ReleaseIfRef();
}

mxt result CExampleTwo::NonDelegatingQueryIf(IN mxt iid iidRequested, OUT void**
          ppInterface)
{
    if(IsEqualEComIID(iidRequested, IID IExampleOne))
    {
        // this interface is offered directly by this object
        *ppInterface = static cast<IExampleOne*>(this);
    }
    else if(IsEqualEComIID(iidRequested, IID IExampleTwo))
    {
        // this interface is offered directly by this object
        *ppInterface = static_cast<IExampleTwo*>(this);
```

```
    }
    else
    {
        return CEComDelegatingUnknown::NonDelegatingQueryIf(iidRequested, ppInterface);
    }
    static cast<IEComUnknown*>(*ppInterface)->AddIfRef();
    return resS OK;
}

//============================================================================
//====  IExampleOne  =========================================================
//============================================================================
void CExampleTwo::IExampleOneMethod()
{
    // Use the inner's object interface to implement the outer object's interface.
    IExampleOne* pIExampleOne = NULL;
    m pInnerIEComUnknown->QueryIf(pIExampleOne);
    if (pIExampleOne != NULL)
    {
        pIExampleOne->IExampleOneMethod();
        pIExampleOne->ReleaseIfRef();
        pIExampleOne = NULL;
    }
}

//============================================================================
//====  IExampleTwo  =========================================================
//============================================================================
void CExampleTwo::IExampleTwoMethod()
{
    printf("IExampleTwoMethod() called\n");
}
```

Code Sample 8 – CExampleTwo Containment Example

Note that **CExampleTwo** implements both **IExampleOne** and **IExampleTwo** interfaces and contains an inner object **m_pInnerIEComUnknown** that is a **CExampleOne** ECOM. The inner object's **IExampleOneMethod** method is used to implement the method **CExampleTwo::IExampleOneMethod** by forwarding the call to the inner object.

Aggregation can be used instead of containment when a component implements an interface exactly as it needs to be used. Instead of re-implementing the inner component's interface when the client queries the outer component for an interface, the outer component queries the inner component for its interface and passes the interface pointer to the client.

Here is an example on how to use aggregation to implement CExampleTwo.

```
class CExampleTwo :    public CEComDelegatingUnknown,
                            public IExampleTwo
{
// Published Interface
//-------------------------
public:
    // << Constructors / Destructors >>
    //-----------------------------------------

    // The constructor will always have one parameter as shown below
    CExampleTwo(IEComUnknown* pOuterIEComUnknown = NULL);
    virtual ~CExampleTwo();

    // << Stereotype >>
    //-----------------------------------------
    static mxt result CreateInstance( IN IEComUnknown* pOuterIEComUnknown,
                                      OUT CEComUnknown** ppCEComUnknown);

    // The following statement is essential, it provides the default and unique
```

```
    // implementation of the IEComUnknown interface from which every other interface
    // inherits.
    MX DECLARE DELEGATING IECOMUNKNOWN

    //-----------------------------------------
    //  Override CEComDelegatingUnknown
    //-----------------------------------------
    virtual mxt result InitializeInstance();
protected:
    virtual void UninitializeInstance(OUT bool* pbDeleteThis);

public:
    virtual mxt result NonDelegatingQueryIf(IN mxt iid iidRequested, OUT void**
          ppInterface);

    //-----------------------------------------
    //  Override IExampleTwo interface method(s)
    //-----------------------------------------
    virtual void IExampleTwoMethod();

// Hidden Methods
//-------------------------
protected:
private:

    // Deactivated Constructors / Destructors / Operators
    //-------------------------------------------------
    CExampleTwo(const CExampleTwo& from);
    CExampleTwo& operator=(const CExampleTwo& from);

// Hidden Data Members
//-------------------------
protected:
private:

    // We must keep a pointer to the object we are aggregating
    IEComUnknown* m pInnerIEComUnknown;
};

mxt result CExampleTwo::CreateInstance(IN IEComUnknown* pOuterIEComUnknown,
                                       OUT CEComUnknown** ppCEComUnknown)
{
    *ppCEComUnknown = MX NEW(CExampleTwo)(pOuterIEComUnknown);
    if ( *ppCEComUnknown == NULL )
    {
        return resFE OUT OF MEMORY;
    }
    return resS OK;
}

//============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ==============================================
//============================================================================
CExampleTwo::CExampleTwo(IEComUnknown* pOuterIEComUnknown):
    CEComDelegatingUnknown(pOuterIEComUnknown),
    m pInnerIEComUnknown(NULL)
{
}

CExampleTwo::~CExampleTwo()
{
}
```

```
//============================================================================
//====  CEComUnknown  ========================================================
//============================================================================
mxt result CExampleTwo::InitializeInstance()
{
    // Create inner component
    mxt result result = CreateEComInstance( CLSID CExampleOne,
                                            GetOwnerIEComUnknown(),
                                            IID IEComUnknown,
                                            reinterpret cast<void**>(&m pInnerIEComUnknown)
          );

    // Validate the creation
    MX ASSERT( MX RIS S(result) && m pInnerIEComUnknown != NULL);
    return result;
}

void CExampleTwo::UninitializeInstance( OUT bool* pbDeleteThis )
{
    CEComUnknown::UninitializeInstance( pbDeleteThis );

    // Release our inner object so it is deleted from memory
    m pInnerIEComUnknown->ReleaseIfRef();

    // add extra destruction steps here
}

mxt result CExampleTwo::NonDelegatingQueryIf(IN mxt iid iidRequested, OUT void**
          ppInterface)
{
    if(IsEqualEComIID(iidRequested, IID IExampleOne))
    {
        // This interface is offered through aggregation --> use inner object we kept
        return m pInnerIEComUnknown->QueryIf(iidRequested,ppInterface);
    }
    else if(IsEqualEComIID(iidRequested, IID IExampleTwo))
    {
        // this interface is offered directly by this object
        *ppInterface = static cast<IExampleTwo*>(this);
    }
    else
    {
        return CEComDelegatingUnknown::NonDelegatingQueryIf(iidRequested, ppInterface);
    }
    static cast<IEComUnknown*>(*ppInterface)->AddIfRef();
    return resS OK;
}

//============================================================================
//====  IExampleTwo  =========================================================
//============================================================================
void CExampleTwo::IExampleTwoMethod()
{
    printf("IExampleTwoMethod() called\n");
}
```

Code Sample 9 – CExampleTwo Aggregation Example

Note **CExampleTwo** does not implement **IExampleOne**. Instead, in **CExampleTwo::NonDelegatingQueryIf** if the interface queried is **IExampleOne**, the inner object **m_pInnerIEComUnknown** is queried for this interface and the interface pointer is returned to the client. Also note that in **CExampleTwo::InitializeInstance**, a pointer to the owner **IEComUnknown** is given to the ECOM factory. When using aggregation, this is particularly important because it indicates to the inner ECOM that it is being aggregated and by whom. An ECOM inner component must know it is being aggregated because it must have the same **IEComUnknown** interface pointer as the outer component to satisfy one of the **QueryIf** rules (“You always get the same

IEComUnknown"). From the outside, the client of the outer component does not know that the outer component aggregates the inner component and does not have access to the inner component's **IEComUnknown**.

6. CServicingThread and CEventDriven Classes

The **CServicingThread** and **CEventDriven** classes are used to help the user with the task of concurrent task programming. Usually, when writing multi-threaded code, a programmer has to take care of many things such as serializing access to certain resources, all while trying to avoid synchronization issues that can arise from such code. The **CServicingThread** and **CEventDriven** alleviate these issues by providing an execution context in which a set of tasks can execute.

6.1 CServicingThread

6.1.1 Introduction

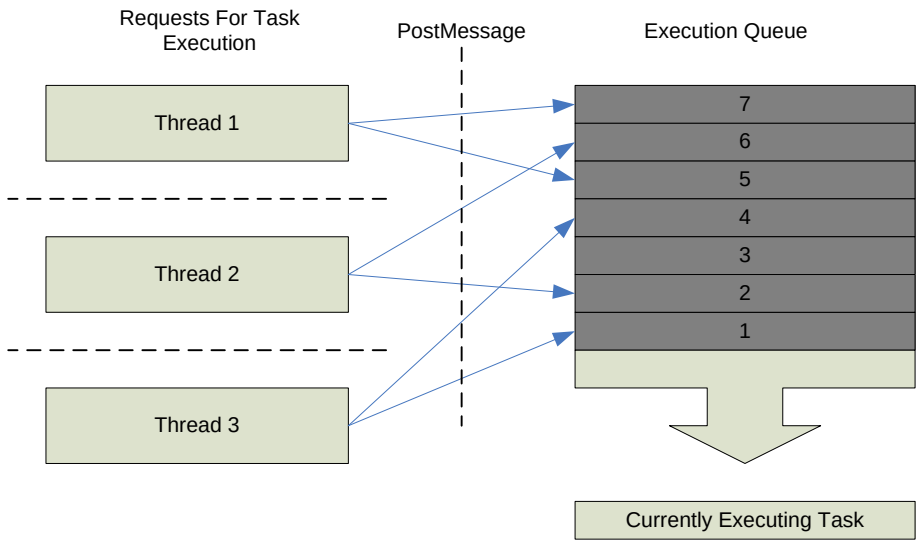
The **CServicingThread** class is used to perform services requested by the user. The code executing in the context of the servicing thread is not controlled directly by the user. Instead, the user requests to the servicing thread that some code be executed in its context. There are three types of user requests that may be serialized into the servicing thread execution context: Messages, Timers and Socket Events.

The **CServicingThread** provides several advantages like the reduction of the number of threads needed in a system. By sharing an instance of **CServicingThread**, it is possible to reduce considerably the number of context switches happening in a system.

6.1.2 Message Service

To implement the message service mechanism, the **CServicingThread** maintains a queue of tasks to be executed in the context of the servicing thread. When a new request is scheduled for execution, it is added at the end of a queue and is executed once all the tasks that were in the queue before the new request have been executed.

Figure 4 - Servicing Thread Queuing Mechanism



6.1.2.1 IMessageServiceMgr

The **IMessageServiceMgr** interface is a very simple interface that must be implemented by the class using the **CServicingThread** message service. It has only one method that is called when a requested task needs to be executed.

Usually, the implementation of this method simply dispatches the message received in the **uMessage** parameter by calling the appropriate internal method.

It is very important to understand that the performance of the **CServicingThread** is directly related to the rapidity of the managers at processing their events. An implementation of the **IMessageServiceMgr::EvMessageServiceMgrAwaken** method should never block and should perform its processing as quickly as possible. If blocking is needed, a worker thread should be used instead.

```
void CNetworkService::EvMessageServiceMgrAwaken(IN bool bWaitingCompletion,
                                                IN unsigned int uMessage,
                                                IN CMarshaler* pParameter)
{
    switch (uMessage)
    {
        case eMSG SEND PACKET:
        {
            InternalSendPacketA(pParameter);
            break;
        }
        case eMSG RECV PACKET:
        {
            InternalRecvPacket(pParameter);
            break;
        }
        default:
        {
            MX ASSERT EX(false, "Received Unknown Task Execution Request");
            break;
        }
    }
}
```

Code Sample 10 - IMessageServiceMgr::EvMessageServiceMgrAwaken Example

6.1.2.2 IMessageService

The **IMessageService** interface of the **CServicingThread** provides the most basic access to the synchronized task execution services of the **CServicingThread** class. This interface provides a direct access to an execution queue. Tasks can be added to this queue for sequential execution from the only method provided by the **IMessageService** interface:

```
virtual mxt result PostMessage(IN IMessageServiceMgr* pManager,
                               IN bool bWaitCompletion,
                               IN unsigned int uMessageId,
                               IN TO CMarshaler* pParameter = NULL) = 0;
```

Code Sample 11 - IMessageService::PostMessage Signature

The user of the **IMessageService** class calls the **PostMessage** method when a task needs to be added on the execution queue. When making the call, the user of the service first provides a manager to the **PostMessage** method letting the **CServicingThread** know who should be contacted when a task is ready for execution.

The second parameter allows the caller of the method to wait for the requested task to finish executing before continuing. By definition, this renders the call to **PostMessage** synchronous and all the currently queued tasks will have to complete in addition of the requested task before **PostMessage** can return.

The last 2 parameters of the **PostMessage** method are used as context and are simply passed back to the manager specified in parameter one. The **uMessageId** parameter contains the message ID so that the manager knows which code to run and the **pParameter** contains a **CMarshaller** reference that contains additional parameters.

The following is a code example of a typical call to **PostMessage**:

```
void CNetworkService::SendPacketA(IN CSocketAddr const* pAddress,
                                  IN bool bRetransmit,
                                  IN IPacketHeader* pPacketHeader,
                                  IN CBlob const* pblobPacketContent
                                  IN SFlags const* pstFlags)
{

    CMarshaler* pParameter = CPool<CMarshaler>::New();

    // Some user-defined types, like the CSocketAddr and CBlob define operators
    // for the insertion into the CMarshaller by value. Some types however do
    // not define this operator and must therefore be passed by reference.
    //
    // The problem with passing parameters to the asynchronous PostMessage
    // method by reference is that we need to guarantee that they will be alive
    // when the message is processed. To acheive this we will make a copy of the
    // pstFlags parameter and transfer its ownership of that copy to the handler
    // for the PostMessage message.

    SFlags* pFlagsCopy = MX NEW(SFlags)(*pstFlags);

    //We need the pPacketHeader ECOM object to be alive when the
    //asynchronous message gets processed. We need to add a reference here
    //and pass the ownership of that reference to the handler for
    //the PostMessage message.
    pPacketHeader->AddIfRef();

    *pParameter << *pAddress;
    *pParameter << bRetransmit;
    *pParameter << pPacketHeader;
    *pParameter << *pblobPacketContent;
    *pParameter << pFlagsCopy;

    m pMessageService->PostMessage(this, false, eMSG SEND PACKET, pParameter);
}
```

Code Sample 12 - IMessageService::PostMessage Usage

There are a few items worth pointing out about the example above:

- The **PostMessage** method takes the ownership of its fourth parameter, which *must* be allocated by using the **CPool<CMarshaller>::New()** method.
- Since method SendPacketA is calling an asynchronous method and passing it parameters, it *must* assure that the parameters will still be alive when the message is processed. It is done by creating copies for parameters that can't be marshaled by value and giving the ownership of those copies to the handler of the requested message.
- The ECOM object passing is handled in a similar manner by making sure that a reference will be available for the trip to the handler.

The above example demonstrates the asynchronous calling model. See the diagram below to see the flow of asynchronous and synchronous calls.

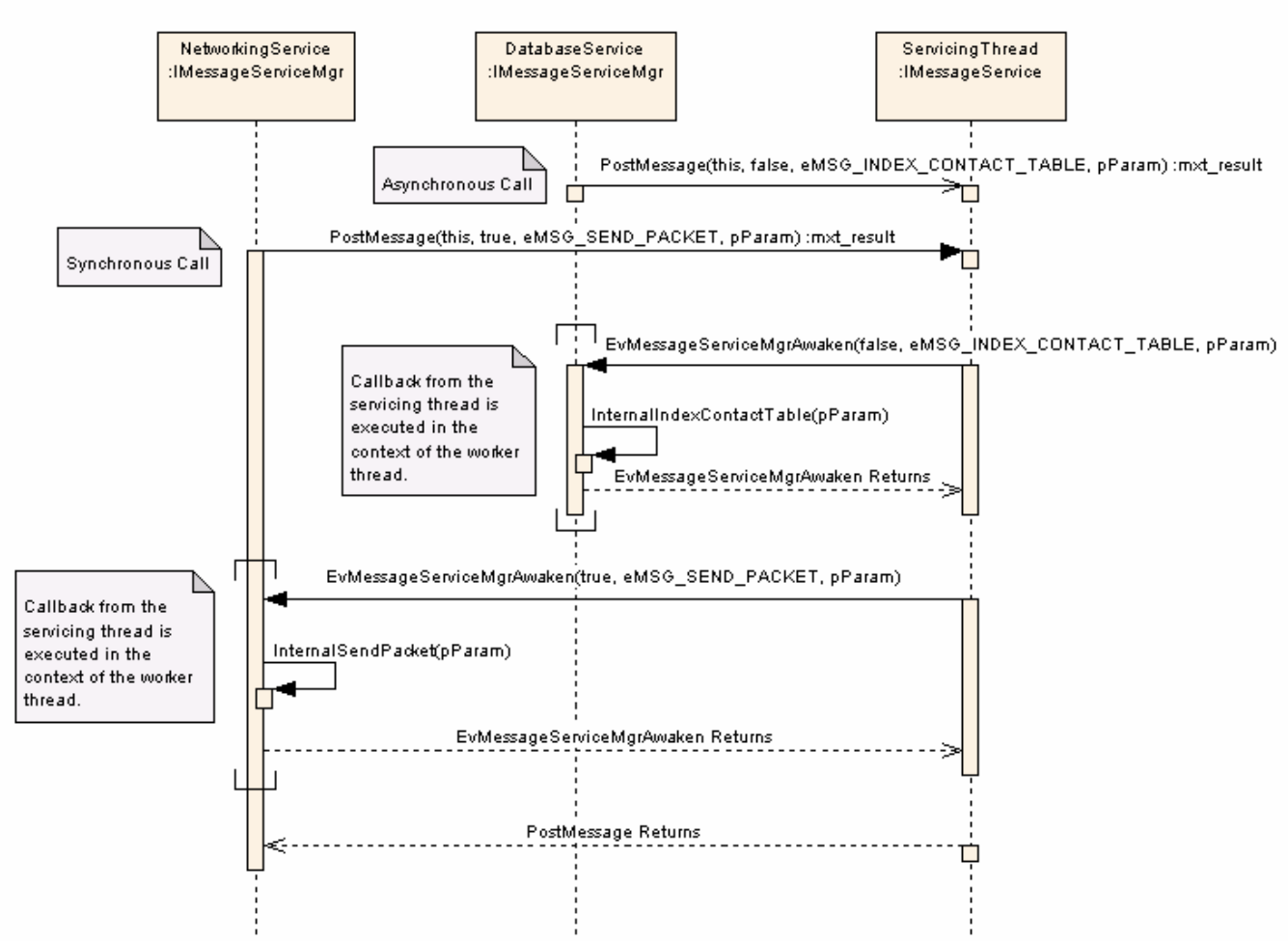


Figure 5 - IMessageService Sequence Diagram

The above diagram shows the two different ways of calling **PostMessage**. The following is a short description of the sequence of events in the example:

1. The **DatabaseService** class first requests the task identified as **eMSG_INDEX_CONTACT_TABLE** to be executed asynchronously. Note that the **PostMessage** call returns immediately before the task has been performed.

2. Before the **ServicingThread** (instance of **CServicingThread**) has had a chance to process the task queue, another request comes in from the **NetworkingService**. This time, the request is synchronous and the **PostMessage** call does not return until the task has been performed.

3. Once the **ServicingThead** has a chance, it processes the first task in the queue. Because the **eMSG_INDEX_CONTACT_TABLE** task was added before the **eMSG_INDEX_CONTACT_TABLE**, it is processed first.

4. The **ServicingThread** sends a signal to the **DatabaseService** that it should perform the **eMSG_INDEX_CONTACT_TABLE** task. The **DatabaseService** does this by calling a private function that performs the task (**InternalIndexContactTable**).

5. When the **InternalIndexContactTable** is finished, the **DatabaseService** returns from the **EvMessageServiceMgrAwaken** call, which allows the servicing thread to move on to the next task.

6. The next task in the queue is the **eMSG_SEND_PACKET** requested synchronously by the **NetworkService** class. This is performed in the same way as the **eMSG_INDEX_CONTACT_TABLE** tasks (via the **IMessageServiceMgr:: EvMessageServiceMgrAwaken** and **InternalSendPacket** call).

7. Once the **eMSG_SEND_PACKET** task has been completed (when the **NetworkService** class returned from the **EvMessageServiceMgrAwaken** call), then the **PostMessage** call for the same task can finally return.

6.1.2.3 Processing Order

One thing that the **CServicingThread** class tries to enforce as much as possible is the order in which requested tasks are processed and sent to the **IMessageServiceMgr** manager. Usually, the order in which the **PostMessage** calls have been made is preserved. This effectively means that if one thread makes multiple calls to **PostMessage**, the **CServicingThread** processes the tasks in the same order as they arrived.

Of course this can only be guaranteed for **PostMessage** calls coming from the same thread. In other words, if multiple threads are each calling **PostMessage** multiple times, then the order of execution is only guaranteed per-thread.

In addition to the above exception in a multi-threading scenario, there is another scenario in which the processing order can differ from the normal case.

```
pMsgService->PostMessage(this, false, eMSG SEND PACKET, pMarshaledSendPacket);
pMsgService->PostMessage(this, true, eMSG_RECV_PACKET, pMarshaledRecvPacket);
```

Code Sample 13 - Message Request Processing Order

Looking at the above code, let's first assume that the **eMSG_SEND_PACKET** message *must* be processed before the **eMSG_RECV_PACKET** for the code to work properly. The **eMSG_SEND_PACKET** message is executed asynchronously to send data over the network. Then the **eMSG_RECV_PACKET** is requested as a synchronous message so that the result of the receive operation can be put as an **OUT** parameter in the **pParameter** parameter (the last parameter) of the **PostMessage** method.

The reason why the **eMSG_SEND_PACKET** message needs to be processed first is because the **eMSG_RECV_PACKET** will be waiting for a response from the party who received the sent packet. If the packet is never sent, then the receive operation will wait forever and will result in a deadlock.

At first sight, the code above looks like it is going to work correctly. In other words **eMSG_SEND_PACKET** should be executed before **eMSG_RECV_PACKET**. However, that is not always true. In fact, the messages are processed in the appropriate order if and only if the **PostMessage** calls come from a thread other than the servicing thread itself.

For a case where the above code is executed from the same thread as the **CServicingThread** thread, this would end up with the following scenario:

1. **PostMessage** is called in an asynchronous manner to request the **eMSG_SEND_PACKET** message and returns right away without executing.

2. **PostMessage** is called in a synchronous manner to request the **eMSG_RECV_PACKET** message and waits until the operation completes.

3. Now, inside the second **PostMessage** there would normally be a **Wait** call made on a synchronization primitive to wait for the operation to complete. However, since it is the same thread as the **CServicingThread**, this would result in a deadlock (i.e. inside the second **PostMessage**, it would be necessary to wait for the thread that was just blocked to process the requested task and **Signal** the completion).

4. When the **CServicingThread** detects that a synchronous request was made from the same thread as its own thread, it deals with the situation by completely by-passing the message queue and performs the synchronous operation right away.

5. Since the **CServicingThread** by-passed the message queue (which contained the **eMSG_SEND_PACKET** message request), it means that the **eMSG_RECV_PACKET** is executed before the **eMSG_SEND_PACKET** and thus causes a deadlock.

To avoid issues like the above example, it is important to understand how the **bWaitForCompletion** parameter works in the case where the **PostMessage** is called from the same thread as the **CServicingThread** itself. The following are some simple tips to help avoid these issues:

- Don't mix asynchronous and synchronous **PostMessage** calls when the order of operation is important.
- Try to avoid making synchronous **PostMessage** requests from the **IMessageServiceMgr::EvMessageServiceMgrAwaken** callback from the **CServicingThread**.
- Transform your synchronous requests into asynchronous ones and return the result of the operation via a manager interface.

6.1.2.4 Coding Example

The code below illustrates a typical usage scenario for the **CServicingThread** messaging service mechanism.

```
#include "MxConfig.h"
#include "FrameworkCfg.h"
#include "CPool.h"
#include "CMarshaller.h"

MX NAMESPACE START(MXD GNS)

//Forward Declaration.
class IDatabaseServiceMgr;

class CDatabaseService : public IMessageServiceMgr
{
public:
    CDatabaseService()
    :   m pMgr(NULL)
    {
    }

    ~CDatabaseService()
    {
        if (m pMessageService)
        {
            m_pMessageService->ReleaseIfRef();
        }
    }

    void Activate(IN IDatabaseServiceMgr* pMgr, IN IEComUnknown* pServicingThread)
    {
        m_pMgr = pMgr;

        //Since we only need the messaging services of the servicing thread, let's
        //acquire that interface now and store it for convenience. Note that this
        //will increment the reference count of the servicing thread.
        pServicingThread->QueryIf(&m pMessageService);
    }

    //This method will fetch customer information in an asynchronous manner.
    //The manager of the CDatabase service class will be notified with the
    //results when the operation completes.
    void FetchCustomerInformationA(IN uint32 t uCustomerId)
    {
        CMarshaler* pParameter = CPool<CMarshaler>::New();

        *pParameter << uCustomerId;

        m pMessageService->PostMessage(this,
                                       false,
                                       eMSG FETCH CUSTOMER INFORMATION,
                                       pParameter);
```

```
    }

    //This method will fetch the inventory count of a certain product in a
    //synchronous manner. The result will be returned by the method.
    uint32 t GetProductInventoryCount(IN uint32 t uProductId)
    {
        CMarshaler* pParameter = CPool<CMarshaler>::New();
        uint32 t uInventoryCountResult = 0;

        *pParameter << uProductId;
        *pParameter << &uInventoryCountResult;

        m pMessageService->PostMessage(this,
                                       true,
                                       eMSG FETCH PRODUCT INVENTORY COUNT,
                                       pParameter);

        //Note that because we called PostMessage and asked that the servicing
        //thread wait for completion of the operation before returning that the
        //uInventoryCountResult will have had time to be updated.
        return uInventoryCountResult;
    }

// Hidden Methods
//----------------
private:
    virtual void EvMessageServiceMgrAwaken(IN bool bWaitingCompletion,
                                           IN unsigned int uMessageId,
                                           IN CMarshaler* pParameter)
    {
        switch (uMessageId)
        {
            case eMSG FETCH CUSTOMER INFORMATION:
            {
                InternalFetchCustomerInformationA(pParameter);
                break;
            }

            case eMSG FETCH PRODUCT INVENTORY COUNT:
            {
                InternalFetchProductInventoryCount(pParameter);
                break;
            }

            default:
            {
                MX ASSERT EX(false, "Invalid Message Id Received");
                break;
            }
        }
    }

    void InternalFetchCustomerInformationA(IN CMarshaler* pParameter)
    {
        unsigned int uCustomerId = 0;
        *pParameter >> uCustomerId;

        ICustomerInfo* pCustomerInfo = NULL;

        // Fetch the customer information from the database and store it in the
        // pCustomerInfo variable.
        DatabaseGetCustomerInfo(uCustomerId, &pCustomerInfo);

        // Once we have the customer information, notify the manager.
        m_pMgr->EvDatabaseServiveMgrFetchedCustomerInformation(uCustomerId,
```

```
                                                                pCustomerInfo);
    }

    void InternalFetchProductInventoryCount(IN CMarshaler* pParameter)
    {
        unsigned int uProductId = 0;
        unsigned int* puProductInventoryCount = NULL;

        *pParameter >> uProductId;
        *pParameter >> puProductInventoryCount;

        // Get the inventory count for the specified product id and store it in
        // the memory pointed to by puProductInventoryCount.
        *puProductInventoryCount = GetInventoryCount(uProductId);
    }

// Hidden Types and Data Members
//------------------------------
private:
    //-- Message Ids for CServicingThread messaging service mechanism.
    //-----------------------------------------------------------------
    enum EMsgId
    {
        eMSG FETCH CUSTOMER INFORMATION = 0,
        eMSG FETCH PRODUCT INVENTORY COUNT,
        //...
    };

    IDatabaseServiceMgr* m pMgr;
    IMessageService* m pMessageService;
};

MX_NAMESPACE_END(MXD_GNS)
```

Code Sample 14 - Messaging Service Example

6.1.3 Timer Service

The timer service is another service provided by the **CServicingThread** class. This service is very similar to the Message Service as it executes requested tasks in the context of the thread owned by the **CServicingThread**. The difference lies in the fact that the requested tasks are performed only after a certain time has elapsed. The timer service also allows for periodic timers, exponential timers, the cancellation of specific timers, or all timers.

6.1.3.1 ITimerServiceMgr

The **ITimerServiceMgr** interface is almost identical to the **IMessageServiceMgr** interface. It is implemented by managers of the **CServicingThread** who wish to use the Timer Services.

The interface has only one method which must be implemented to receive notification that a timer has fired and that the corresponding code must be executed.

It is very important to understand that the performance of the **CServicingThread** is directly related to the rapidity of the managers at processing their events. An implementation of the **ITimerServiceMgr::EvTimerServiceMgrAwaken** method should never block and should perform its processing as quickly as possible. If blocking is needed, a worker thread should be used instead.

```
virtual void EvTimerServiceMgrAwaken(IN bool,
                                     IN unsigned int uTimerId,
                                     IN mxt_opaque opq) = 0;
```

Code Sample 15 - ITimerServiceMgr::EvTimerServiceMgrAwaken Signature

Typical implementation for this method is to switch on the **uTimerId** parameter and perform the action corresponding to this timer. See section *6.1.2.1 IMessageServiceMgr* for an example of how to do this.

If the first parameter of this event is set to true, it means that the timer has been stopped in one of four ways:

1. Manually, when calling the **ITimerService::StopTimer** method for the timer specified with the **uTimerId** parameter.

2. Manually, when calling the **ITimerService::StopAllTimers**, which will trigger the **EvTimerServiceMgrAwaken** for each currently active timer.

3. Automatically, when one of the **StartTimer** methods is called for a currently existing timer, effectively resetting it.

4. Automatically, when a periodic exponential timer set with the **bStopAtCeiling** to true reaches the ceiling timeout value set with the **uCeilingTimeoutMs**.

The only other major difference between the **IMessageServiceMgr** and the **ITimerServiceMgr** is the last parameter of their **Awaken** method. For the timer, an opaque value is passed instead of a **CMarshaller** pointer, when starting the timer and then each time the timer fires, the **CServicingThread** sends this opaque back with the **EvTimerServiceMgrAwaken** event. This opaque can be used to convey anything to the callback of the timer, but it is up to the manager class to manage the lifetime of whatever object might have been passed under the umbrella of the opaque value.

6.1.3.2 ITimerService

The **ITimerService** interface provides access to the timer services of the **CServicingThread** class. This interface is used to manipulate timers that trigger code to be executed in the context of the **CServicingThread**. Each timer is created by using one of the two overloads of the **ITimerService::StartTimer** method.

```
virtual mxt_result StartTimer(IN ITimerServiceMgr* pManager,
                              IN unsigned int uTimerId,
                              IN uint64_t uFloorTimeoutMs,
                              IN uint64_t uCeilingTimeoutMs,
                              IN unsigned int uMultBy,
                              IN unsigned int uDivBy,
                              IN bool bStopAtCeiling,
                              IN mxt_opaque opq = 0,
                              IN EPeriodicity ePeriodicity) = 0;

virtual mxt_result StartTimer(IN ITimerServiceMgr* pManager,
                              IN unsigned int uTimerId,
                              IN uint64_t uTimeoutMs,
                              IN mxt_opaque opq = 0,
                              IN EPeriodicity ePeriodicity) = 0;
```

Code Sample 16 - ITimerService::StartTimer Signature

The first version of the **StartTimer** method starts an exponential timer. This timer *must* be periodic and is triggered at exponentially increasing intervals.

The second version starts a normal timer. This timer can be a one-shot timer or a constant periodic timer that will continue to be triggered until stopped.

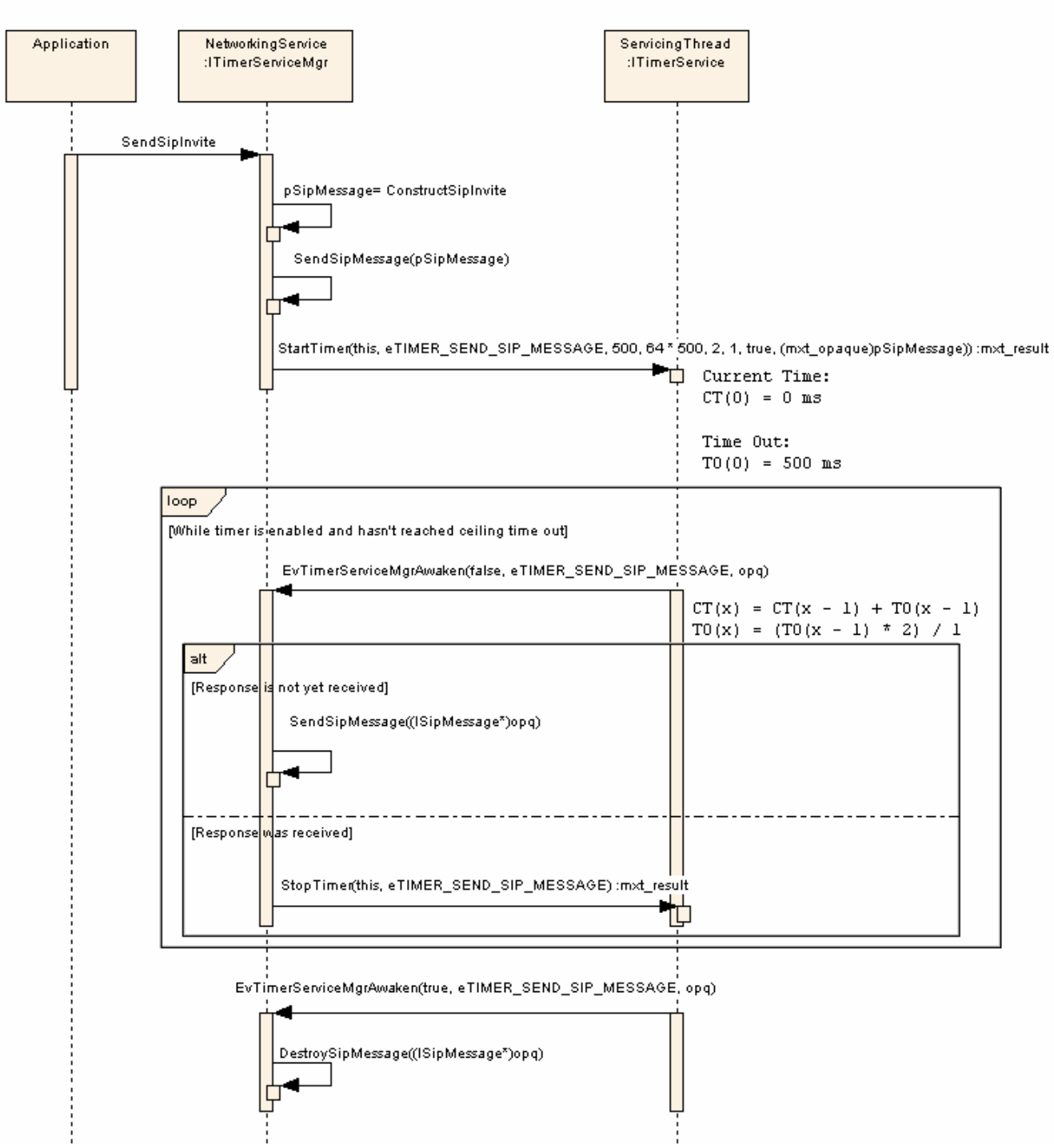
To stop a timer, the **StopTimer** or **StopAllTimers** methods are used.

6.1.3.3 Example

Figure 6 - Exponential Timer Sequence Diagram

The above figure shows a practical example using an exponential timer. In the example, the application requests that a **SIP INVITE** be sent by the **NetworkingService** class. According to the **SIP** RFC, this needs to be done at exponentially increasing intervals until a response from the remote party is received. The following are the steps to achieve this:

1. When the **NetworkingService** class receives the request to send a **SIP** message, it constructs this message and sends it. Once this has been done the exponential timer is started. The formula for determining when the timer will fire is as follows:



$T_0(x) = T_0 + T_f * (m / d)^x$, Where $TO(x) < T_c$ and $x = 0, 1, 2, 3, ...$

Where $\mathbf{T_0}$ is the absolute initial time at which the timer was started, $\mathbf{T_f}$ is the floor timeout corresponding to the **uFloorTimeoutMs** parameter, $\mathbf{T_c}$ is the ceiling timeout corresponding to the **uCeilingTimeoutMs** parameter, **m** is the multiplier corresponding to the **uMultBy** parameter, **d** is the divisor corresponding to the **uDivBy** parameter, and finally, **x** represents the number of times the timer has been triggered.

2. After the timer has been started, the **NetworkingService** class can expect to be notified whenever the timer gets triggered.

3. When the timer gets triggered, the **NetworkingService** checks if a response has been received for the **SIP** message it sent. If a response has not yet been received, the **NetworkingService** re-sends the **SIP** message. This continues at exponential intervals until a response is received or the ceiling timeout (**uCeilingTimeoutMs**) is reached.

4. Once the response is received, the **NetworkingService** no longer needs the timer and as such calls **StopTimer** to stop the timer.

5. After the timer has been stopped, the **CServicingThread** sends a notification that this happened. When the **NetworkingService** receives it, it makes sure to cleanup anything associated with the timer (i.e. the **SIP** message stored in the opaque value).

6.1.3.4 Coding Example

```
#include "ITimerServiceMgr.h"
#include "ITimerService.h"
#include "ISipMessage.h"

MX NAMESPACE START(MXD GNS)

struct SRemotePartyInfo;

class CNetworkingService : public ITimerServiceMgr
{
public:
    CNetworkingService()
    {
    }
    ~CNetworkingService()
    {
        if (m pTimerService)
        {
            // This is a synchronous method, when it returns all timers will
            // have been stopped after having notified their managers.
            m pTimerService->StopAllTimers();

            //Finally, release the timer service.
            m pTimerService->ReleaseIfRef();
        }
    }
    void Activate(IN IEComUnknown* pServicingThread)
    {
        // Since we only need the timer services of the servicing thread, let's
        // acquire that interface now and store it for convenience. Note that
        // this will increment the reference count of the servicing thread.
        pServicingThread->QueryIf(&m pTimerService);
    }

    void SendSipInviteA(IN SRemotePartyInfo const& rRemotePartyInfo)
    {
        ISipMessage* pSipMessage = NULL;

        // Create the SIP INVITE from the SRemotePartyInfo structure.
```

```
        CreateSipInvite(rRemotePartyInfo, &pSipMessage);

        // Initial send of the SIP message.
        SendSipMessage(pSipMessage);

        // Start the timer for sending re-transmits. Note that here we transfer
        // the ownership of the SIP message to the timer mechanism. The
        // pSipMessage will eventually get destroyed in the
        // EvTimerServiceMgrAwaken method.
        m pTimerService->StartTimer(this,
                                    eTIMER SEND SIP MESSAGE,
                                    500,
                                    500 * 64,
                                    2,
                                    1,
                                    true,
                                    MX VOIDPTR TO OPQ(pSipMessage));
    }
// Hidden Methods
//----------------
private:
    virtual void EvTimerServiceMgrAwaken(IN bool bStopped,
                                         IN unsigned int uTimerId,
                                         IN mxt opaque opq)
    {
        switch (uTimerId)
        {
            case eTIMER SEND SIP MESSAGE:
            {
                ISipMessage* pSipMessage =
          reinterpret cast<ISipMessage*>(MX OPQ TO VOIDPTR(opq));
                if (bStopped)
                {
                    //Timer was stopped, time to release the SIP message.
                    pSipMessage->ReleaseIfRef();
                }
                else
                {
                    if (pSipMessage->IsResponseReceived())
                    {
                        m pTimerService->StopTimer(this,
                                                   eTIMER SEND SIP MESSAGE);
                    }
                    else
                    {
                        //Re-send the SIP message.
                        SendSipMessage(pSipMessage)
                    }
                }
                break;
            }
            default:
            {
                MX ASSERT EX(false, "Invalid Timer Id Received");
                break;
            }
        }
    }

    void CreateSipInvite(IN SRemotePartyInfo const& rRemotePartyInfo,
                         OUT ISipMessage** ppSipMessage);

    void SendSipMessage(IN ISipMessage* pSipMessage);

    enum ETimerId
```

```
    {
        //-- Timer Ids for CServicingThread time service mechanism.
        //-----------------------------------------------------------------
        eTIMER SEND SIP MESSAGE = 0
    };

    ITimerService* m pTimerService;
};

MX_NAMESPACE_END(MXD_GNS)
```

Code Sample 17 - Timer Service Coding Example

6.1.4 Socket Service

The socket service of the **CServicingThread** allows the user to monitor sockets for readability, writability, and socket exception events. Like all the above services, the socket service sends these events in the thread context of the **CServicingThread** to allow for easy synchronization.

6.1.4.1 ISocketServiceMgr

Once again, note that the **ISocketServiceMgr** interface is almost identical to the **ITimerServiceMgr** and **IMessageServiceMgr** interfaces. It has one method on which socket events are received in the context of the **CServicingThread**.

It is very important to understand that the performance of the **CServicingThread** is directly related to the rapidity of the managers at processing their events. An implementation of the **ISoketServiceMgr::EvSocketServiceMgrAwaken** method should never block and should perform its processing as quickly as possible. If blocking is needed, a worker thread should be used instead.

```
virtual void EvSocketServiceMgrAwaken(IN mxt hSocket hSocket,
                                      IN unsigned int uEvents,
                                      IN mxt_opaque opq) = 0;
```

Code Sample 18 - ISocketServiceMgr::EvSocketServiceMgrAwaken Signature

The **hSocket** parameter specifies the handle of socket for which the event occurred. The **uEvents** specifies the type of event that has occurred (one of **uSOCKET_READABLE**, **uSOCKET_WRITABLE**, or **uSOCKET_IN_EXCEPTION**). The **opq** parameter contains user data that was passed in at the time that the socket was registered to receive events.

It is important to understand that once an event has been signaled, its detection is automatically disabled by the CServicingThread. The user must call EnableEventDetection to re-enable the detection of the event. This is because it is possible for the user to process the event detection asynchronously.

6.1.4.2 ISocketService

To receive socket notifications, a user of the socket service must use the **ISocketService** interface to register sockets for event monitoring. This is done through the **ISocketService::RegisterSocket**.

```
virtual mxt result RegisterSocket(IN mxt hSocket hSocket,
                                  IN ISocketServiceMgr* pManager,
                                  IN mxt_opaque opq) = 0;
```

Code Sample 19 - ISocketService::RegisterSocket Signature

Once a socket has been registered, the **CServicingThread** monitors it for the events specified by the **ISocketService::EnableEventsDetection** and **ISocketService::DisableEventsDetection** methods. These methods allow the user to choose the type of event that will be monitored by the **CServicingThread** for a specific socket.

The socket event monitoring service of the **CServicingThread** allows the user to serialize calls to sockets with other services in the same thread to allow for easy concurrent network programming. This is achieved by the **CServicingThread** by waiting

for events for a particular socket using the **CPollSocket** class in the **Network** framework package. The **CPollSocket** in turn uses the operating system's **select** function to wait for state changes on the different sockets.

It is worth noting that although the RegisterSocket method seems to be solely targeted for socket management, this is not entirely the case. It is possible on some operating system (e.g. Linux, Solaris, etc.) to pass any file descriptor like pipes for instance.

The user must call **IStocketService::UnregisterSocket** when the socket is about to be closed or when it no longer requires the service of the **CServicingThread**.

```
virtual mxt result UnregisterSocket(IN mxt hSocket hSocket) = 0;
```

Code Sample 20 - ISocketService::UnregisterSocket Signature

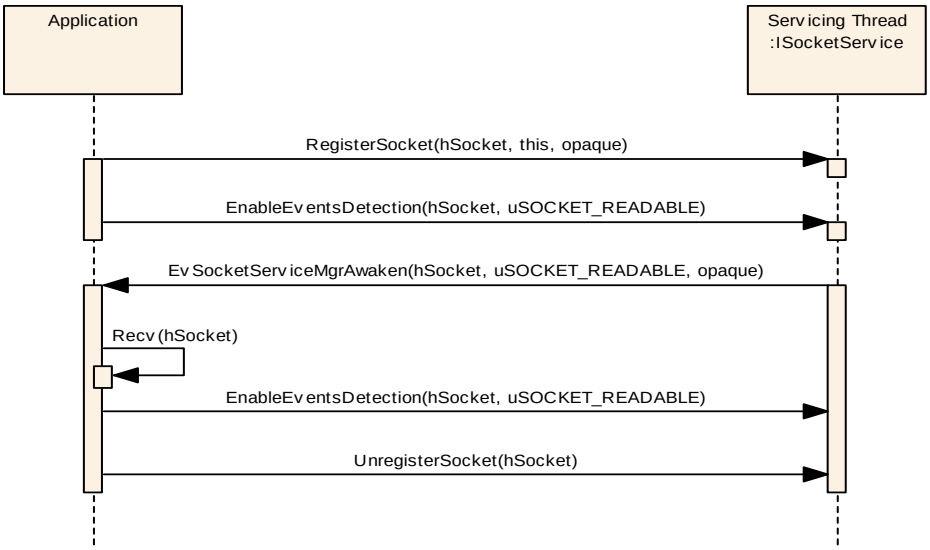
6.1.4.3 Example

Figure 7 - Socket Event Detection Sequence Diagram

The application registers one socket handle and enables the detection of the event uSOCKET_READABLE. Asynchronously, data is received on the socket handle. The servicing thread that was blocked on the select system call unblocks and notifies the registered **ISocketServiceMgr** that the socket handle is now readable. The data is read on the socket handle until no more data is available, after which the event detection is re-enabled to allow the detection of future received data. When the application no longer needs the socket handle, it un-registers it and closes it. If the application does not un-register the socket handle before closing it, the selected call will fail from now on, generating a tight loop in the servicing thread.

6.1.5 Activation Service

Before a class can use the above services of the **CServicingThread**, the **CServicingThread** needs to be activated. The activation is done through the **IActivationService** interface, which creates the context in which requests are processed and it make the **CServicingThread** ready to accept these requests.



6.1.5.1 IActivationService

To activate the **CServicingThread**, the user goes through the **IActivationService** interface. This interface has three methods and two of those methods are overloads of the **IActivationService::Activate** method.

```
virtual mxt_result Activate(IN const char* pszName = NULL,
                            IN uint32_t uStackSize = 0,
                            IN CThread::EPriority ePriority = CThread::eNORMAL) = 0;

virtual mxt_result Activate(IN uint64_t uTimeoutMs,
                            OUT bool* pbReadyToRelease) = 0;
```

Code Sample 21 - IActivationService::Activate Signature

The first overload of the Activate method activates the **CServicingThread** in the context of a new thread that is exclusively owned by the **CServicingThread**. This thread is used to execute all tasks requested to the **CServicingThread** and will run until the last reference to the **CServicingThread** is released. The parameters of this first version are passed directly to the **CThread::StartThread** call that is used internally.

The second version of the Activate method activates the **CServicingThread** in the execution context of the currently executing thread. As such, this function will block until one of the following happens:

1. The timeout specified in **uTimeoutMs** elapses.

2. The reference count for the **CServicingThread** drops to 1 (the one reference necessary to call the **Activate** method).

Upon returning, the **Activate** method fills the output parameter **pbReadyToRelease** with **false** if the first case above occurred first and **true** if the second case occurred first. If **pbReadyToRelease** has the value **true**, it also means that the reference used to make the call to **Activate** is the last reference remaining.

The last method of the **IActivationService** interface is the **IsCurrentExecutionContext** method, which allows the caller to determine if he is currently executing in the context of the **CServicingThread**. This is often useful, such as in the case where a method can be called from many different threads and the method needs to execute in the context of the **CServicingThread**.

```
void CNetworkingService::SetLocalPort(IN uint16_t uLocalPort)
{
    //We have to make sure to be in the right execution context before
    //we set this value otherwise we will run into synchronization problems.
    if (m_pActivationService->IsCurrentExecutionContext())
    {
        m_uLocalPort = uLocalPort;
    }
    else
    {
        CMarshaler* pMarshaller = CPool<CMarshaler>::New();
        *pMarshaller << uLocalPort;

        //We are not in the correct thread to perform the operation
        //without synchronization. Here we will switch context by
        //using the message service of the CServicingThread.
        m_pMessageService->PostMessage(this,
                                       true,
                                       eMSG_SET_LOCAL_PORT,
                                       pParameter);
    }
}

virtual void CNetworkingService::EvMessageServiceMgrAwaken(IN bool bWaitingCompletion,
                                                           IN unsigned int uMessageId,
                                                           IN CMarshaler* pParameter)
{
    switch(uMessageId)
    {
```

```
        case eMSG_SET_LOCAL_PORT:
        {
            uint16_t uLocalPort = 0;

            *pParameter >> uLocalPort;

            //Now we are in the right thread to perform the operation.
            SetLocalPort(uLocalPort);
            break;
        }
        default:
        {
            MX_ASSERT_EX(false, "Invalid Message Id Received");
            break;
        }
    }
}
```

Code Sample 22 - IActivationService::IsCurrentExecutionContext Example

In the example above, you can see the implementation of a method that can be called from any thread. If it is called from a different thread than the **CServicingThread**, a context switch is made so that no synchronization issues arise.

6.1.5.2 Code Sample

The code for a typical application of the activation with an internal thread is quite trivial. The only restriction involved is the fact that the **Activate** method must be called before any other services are used.

This restriction also applies for the second version of **Activate** but the difference lies in the fact that the second version only processes events while the call to **Activate** is processing. The following is an example of the usage of the second version of the **Activate** method.

```
class CConnectionServer
{
public:
    void RegisterConnection(IN IConnection* pConnection)
    {
        //Make sure to increase the ref-count before adding in the tree.
        pConnection->AddIfRef();
        m_treeConnectionList.Insert(pConnection);
    }

    void UnregisterConnection(IN IConnection* pConnection)
    {
        m_treeConnectionList.EraseElement(pConnection)
        //Make sure to release the connection once we are done.
        pConnection->ReleaseIfRef();
    }

    void Run()
    {
        // A list of released connections.
        CList<IConnection*> listReleasedConnections;
        while (IsRunning())
        {
            // Go through the list of connections and allocate them each a 20ms
            // interval of activation.
            unsigned int uSize = m_treeConnectionList.GetSize()
            unsigned int uIndex = 0;
            for (;uIndex < uSize; ++uIndex)
            {
                IConnection* pConnection = m_treeConnectionList.GetAt(uIndex);
```

```
                //The connection class contains a CServicingThread through which
                //it performs all its tasks.
                IActivationService* pActivationService = NULL;
                pConnection->GetServicingThread(&pActivationService);

                bool bReadyToRelease = false;

                // Here we allocate a slot of time to each registered and active
                // connection.
                pActivationService->Activate(20, &bReadyToRelease);
                if (bReadyToRelease)
                {
                    listReleasedConnections.Append(pConnection)
                }
            }

            while(listReleasedConnections.GetSize() > 0)
            {
                //Get the first released connection.
                IConnection* pReleasedConnection = listReleasedConnections.GetAt(0);

                UnregisterConnection(pReleasedConnection);

                //Now remove it from out list of released connections.
                listReleasedConnections.Erase(0);
            }
        }
    }

private:
    CAATree<IConnection*> m treeConnectionList;
};
```

Code Sample 23 - External Thread Activation Example

6.2 CEventDriven

The **CEventDriven** class is a helper class to facilitate the use of the **CServicingThread** class in many typical cases. The most important issue solved by **CEventDriven** is the ordered destruction of code that uses the **CServicingThread**.

The **CEventDriven** class is meant to be derived from by classes that wish to use any of the services provided by the **CServicingThread** class. As such, the **CEventDriven** derives from all the manager interfaces of the **CServicingThread** and abstracts the service interfaces into member functions so that no **ECOM** handling is necessary.

The **CEventDriven** class has many public methods. Most of these methods are variants of the methods defined by the four interfaces implemented by the **CServicingThread** class. The only difference between the interface methods and the **CEventDriven** methods is that the latter never take any managers as parameters. Instead, the **CEventDriven** object itself is always used as the manager.

6.2.1 Activation Mechanism

The activation mechanism of a CEventDriven object is controlled by one method only:

```
// Summary:
//  Must be called to associate a Servicing Thread with the Event Driven.
//-------------------------------------------------------------------------
mxt_result Activate(IN IEComUnknown* pIEComUnknown);
```

Code Sample 24 - CEventDriven::Activate Signature

The **Activate** method takes a pointer to a **CServicingThread** instance as only parameter. If a valid instance of a **CServicingThread** is passed, then the **CEventDriven** class uses that instance as its servicing thread. Note that when this is

the case, the caller of the **Activate** method is responsible for manually activating the passed-in **CServicingThread** instance. By calling this method, it is effectively possible to share a single **CServicingThread** among multiple **CEventDriven**, which reduces considerably the amount of thread and context switches in a system.

Another way of activating the **CEventDriven** object is by passing **NULL** to the **Activate** method. When the **CEventDriven** object is activated in this way, a new **CServicingThread** is automatically created and activated with default values. This forces the creation of a new thread in the system.

6.2.2 Release Mechanism

One of the delicate parts about working with the **CServicingThread** is the cleanup of the manager classes that have currently pending requests with a **CServicingThread**.

Consider the scenario where multiple timers, socket events, or messages have been queued for execution in the **CServicingThread** context through the managers for these respective events. Then consider that one of the managers needs to be destroyed. The problem here is that if the manager is destroyed before some of the messages or events get sent from the **CServicingThread**, this results in an undefined behaviour.

To get around this issue, the **CEventDriven** class provides methods and mechanisms for releasing objects that are derived from it. This mechanism is mostly abstracted to the user but certain methods can also be overridden to provide flexibility when it is necessary.

```
// Summary:
//  Releases an Event Driven.
//----------------------------
void Release();

// Summary:
//  Finalizes and releases an Event Driven.
//----------------------------------------
void FinalizeAndReleaseA();
```

Code Sample 25 - CEventDriven Release Mechanism Methods

Calling one of the two methods above is the only valid way of properly releasing a **CEventDriven** object. If this rule is not followed, the program behaviour is undefined.

The two release mechanisms will eventually release the **CEventDriven** object. However, this will not be done immediately because there might still be messages remaining to be processed by the **CServicingThread** managed by the **CEventDriven** object. As such, the release mechanism will wait until no messages are left to be processed before releasing the **CEventDriven** object. The release methods above provide two different ways of doing this:

The **CEventDriven::Release** starts the release mechanism and sets a flag indicating that the object should be silent from now on. The **CEventDriven::FinalizeAndReleaseA** starts the release mechanism in a similar way but does not set the silent flag.

The silent flag can be retrieved by the users of the **CEventDriven** object by using the **CEventDriven::IsSilent** method.

The responsibility of ensuring that the object is silent falls on the class derived from the **CEventDriven** class. This is achieved by calling the **CEventDriven::IsSilent** method just before sending any notifications. **CEventDriven::IsSilent** may also be called to discover if the processing unrelated to the destruction of the object can be skipped.

It is very important to understand that once the call to **CEventDriven::Release** returns, the object is considered released by the caller of the method. The object may in fact be released or it may not when the method returns. More importantly, past that point, the object will never manifest again and will finish destroying itself silently.

The **CEventDriven::Release** method performs different processing depending on whether or not the caller thread is the same as the **CServicingThread**. If it is the same, **CEventDriven::Release** starts the destruction of the object asynchronously. If it is called from a different thread, **CEventDriven::Release** is processed synchronously and only returns once CEvenDriven has called the virtual method **CEventDriven::ReleaseInstance**, which normally performs a MX_DELETE(this).

6.2.3 Override Mechanism

As stated above (see section *6.2 CEventDriven*), the **CEventDriven** class derives from all of the manager interfaces for all of the **CServicingThread** services. The reason for this is that the **CEventDriven** class is the manager for all the services of the **CServicingThread** it contains.

This allows the **CEventDriven** class to simplify the handling of the **CServicingThread** services and to remove the need to specify a manager when making a call to **CEventDriven::PostMessage**, **CEventDriven::RegisterSocket**, or **CEventDriven::StartTimer**.

As such, any class deriving from **CEventDriven** wishing to receive events from the **CServicingThread** services need to override these methods. The following is an example with the messaging service:

```
class CNetworkingService : public CEventDriven
{
private:

    enum EMsgId
    {
        eMSG_SEND_SIP_MESSAGE = 0,
        eMSG SEND PACKET
        // ... and so on
    };

    virtual void EvMessageServiceMgrAwaken(IN bool bWaitingCompletion,
                                           IN unsigned int uMessage,
                                           IN CMarshaler* pParameter)
    {
        switch(uMessage)
        {
            case eMSG SEND SIP MESSAGE:
            {
                // Process private message...
            }
            case eMSG_SEND_PACKET:
            {
                // Process private message...
            }
            default:
            {
                // Give a chance to the base class to process it's messages
                CEventDriven::EvMessageServiceMgrAwaken(bWaitingCompletion,
                                                        uMessage,
                                                        pParameter);
            }
        }
    }

    void EvSocketServiceMgrAwaken(IN mxt hSocket hSocket,
                                  IN unsigned int uEvents,
                                  IN mxt opaque opq)
    {
        // Implementation ...
    }

    void EvTimerServiceMgrAwaken(IN bool bStopped,
                                 IN unsigned int uTimer,
                                 IN mxt opaque opq)
    {
        // Implementation ...
    }
};
```

Code Sample 26 - CEventDriven Events Capturing

CAUTION!

In the above example, you can notice that an implementation skeleton is given for the **EvMessageServiceMgrAwaken** notification, while none is given for the other. This is intentional.

The skeleton, as you can see above, is necessary for the correct functioning of the **CEventDriven** class. The latter uses the messaging service internally to implement the release mechanism. As such, it is REQUIRED that the **CEventDriven::EvMessageServiceMgrAwaken** base implementation be called for all unhandled messages.

There is one other restriction related to the fact that the **CEventDriven** class uses the messaging services internally. This restriction limits the choice of message IDs that can be used by a derived class to the range:

0 <= Message Id <= 2^{32} - 3

6.2.4 CEventDriven Example

This example illustrates the use of the release mechanism, the difference between **CEventDriven::Release** and **CEventDriven::FinalizeAndReleaseA**, the use of the **CEventDriven::IsSilent** method, and the overriding of **CEventDriven** protected methods to capture events from the **CServicingThread** services.

Let's start by taking the **CNetworkingService** class from some of the above examples and add to it some functionalities so it becomes something like this:

```
class CNetworkingService : public CEventDriven
{
    // This method will send an INVITE to the specified remote party. When
    // the INVITE is answered or fails, it will notify the specified
    // ISipServiceMgr.
    void SendSipInviteA(IN ISipServiceMgr* pMgr,
                        IN SRemotePartyInfo const& rRemotePartyInfo);

private:

    // CEventDriven:: EvSocketServiceMgrAwaken protected method override.
    virtual void EvSocketServiceMgrAwaken(IN mxt hSocket hSocket,
                                          IN unsigned int uEvents,
                                          IN mxt opaque opq)
    {
        if (uEvents == uSOCKET READABLE)
        {
            // The opq stores the manager, but we don't know yet what type.
            // The InternalSocketReadyForRead method will know, so just pass it
            // blindly.
            InternalSocketReadyForRead(hSocket, opq);
        }
    }

    void InternalSocketReadyForRead(IN mxt hSocket hSocket, IN mxt opaque opq)
    {
        if (m sockUdpSipSocket->GetHandle() == hSocket)
        {
            ISipTransaction* pSipTransaction = NULL;
            ProcessIncomingMessage(hSocket, &pSipTransaction);

            //This is a SIP message, the manager is therefore a ISipServiceMgr*
            ISipServiceMgr* pSipServiceMgr =
                reinterpret cast<ISipServiceMgr*>(MX OPQ TO VOIDPTR(opq));

            if (pSipTransaction->IsOutgoing())
            {
                switch(pSipTransaction->GetType())
                {
                    // ...
                    case eSIP_TRANSACTION_TYPE_INVITE:
```

```
                    {
                        switch(pSipTransaction->GetResponseCode())
                        {

                            // ...

                            case 200: // Received 200 OK
                            {
                                // Make sure that we are not supposed to be silent
                                // before sending the notification.
                                if (!IsSilent())
                                {
                                    pSipServiceMgr->EvSipServiceMgrCallAnswered(
                                        pSipTransaction->GetRemotePartyInfo());
                                }
                                break;
                            }
                            default:
                            {
                                // Make sure that we are not supposed to be silent
                                // before sending the notification.
                                if (!IsSilent())
                                {
                                    pSipServiceMgr->EvSipServiceMgrError(
                                        pSipTransaction->GetResponseCode());
                                }
                                break;
                            }
                        }
                    }
                    default:
                    {
                        MX_ASSERT_EX(false, "Invalid SIP transaction type");
                    }
                }
            }
        }
    }
};
```

Code Sample 27 - CNetworkingService with Some SIP Processing Added

You can see in the above code sample the new **SIP** processing that is added to the **CNetworkingService** class. This new processing requires the user of the **CNetworkingService::SendSipInviteA** to implement a manager interface on which events are sent.

Now consider an **Application** class that is using this new **CNetworkingService** class:

```
class Application : public ISipServiceMgr
{
public:
    void Initialize()
    {
        m_pNetworkingService = MX_NEW(CNetworkingService)();
    }
    void Finalize();

    void MakeCall(IN SRemotePartyInfo const& rRemoteParty)
    {
        // This call will execute asynchronously and when the call is established
        // we will receive an event from the CSipService class.
        m pNetworkingService->SendSipInviteA(this, rRemoteParty);
    }
private:
```

```
    void EvSipServiceMgrCallAnswered(IN SRemotePartyInfo const& rRemoteParty)
    {
        // Show UI for call ...
    }
    void EvSipServiceMgrError(IN uint32 t uResponseCode)
    {
        // Log error ...
    }

    // Private data members
    CNetworkingService* m pNetworkingService;
};
```

Code Sample 28 - Application Class Using SIP Services of CNetworkingService

You can see in the above code sample the simple use of the **CNetworkingService::SendSipInviteA** and the events that it can generate.

Now, as you can see, the **Application** class is missing a release mechanism (i.e. implementation of **Application::Finalize**). Let's first add a release mechanism using the **CEventDriven::Release** to clean up the **CNetworkingService** instance:

```
void Application::Finalize()
{
    m_pNetworkingService->Release();

    // Make sure that no more calls can be made to the networking service.
    m_pNetworkingService = NULL;

    // Now that we have called CEventDriven::Release, we know that the object
    // will be silent. Therefore we are able to destroy the Application class
    // right away without fear that an event will be sent from the m pNetworkingService
    // which is in the process of being released.
    delete this;
}
```

Code Sample 29 - Application::Finalize using CEventDriven::Release

You can see from the above code sample that as soon as **CEventDriven::Release** is called, it can be assumed that the object is dead and will not be sending any events. In reality, the object is working hard to release itself but it may take some time to do so.

The second implementation of **Application::Finalize** uses **CEventDrivent::FinalizeAndReleaseA**, which allows the Application class to process all remaining events generated by the remaining tasks in the **CServicingThread** of the **CEventDriven** object.

```
void Application::Finalize()
{
    m pNetworkingService->FinalizeAndReleaseA();
    // Make sure that no more calls can be made to the networking service.
    m pNetworkingService = NULL;

    // Since we have called FinalizeAndReleaseA, we can't assume that the
    // m pNetworkingService instance will not call us back with an event. As such,
    // we can't release the application right away. Set a flag so that we know
    // that a finalize was requested and is being deferred.
    m bIsFinalizing = true;
}
```

Code Sample 30 - Application::Finalize Using CEventDriven::FinalizeAndReleaseA

In the last implementation of **Application::Finalize,** you can see that the application is not fully finalized. To allow you to complete the finalization, you need to add some modifications to the **CNetworkingService** and **Application** classes to allow the release mechanism to work completely.

```
class CNetworkingService : public CEventDriven
{
public:

    // Constructor which takes a manager.
    void CNetworkingService(IN INetworkingServiceMgr* pMgr)
    :   m pNetworkingServiceMgr(pMgr)
    {
    }

    // ...

private:

    // ...

    // CEventDriven::ReleaseInstance protected method override that gets called
    // when the object is ultimately released.
    virtual void ReleaseInstance()
    {
        //Notify our manager that we are truly released.
        m pNetworkingServiceMgr->EvNetworkingServiceMgrReleased();

        //Call the base class to actually perform the release.
        CEventDriven::ReleaseInstance();
    }

    INetworkingServiceMgr* m pNetworkingServiceMgr;

    // ...
};

class Application : public ISipServiceMgr, public INetworkingServiceMgr
{

private:

    // INetworkingServiceMgr implementation
    virtual void EvNetworkingServiceMgrReleased()
    {
        // The NetworkingService is telling us that it is being released. If
        // a Finalize was deferred, make sure to process it here.
        if (m bIsFinalizing)
        {
            // It is now safe to delete the Application class.
            delete this;
        }
    }
};
```

Code Sample 31 - Additional Framework Necessary to Use CEventDriven::FinalizeAndReleaseA

6.2.5 CEventDriven::ReleaseInstance

It has been mentioned previously that **CEventDriven** provides methods and mechanisms to help releasing objects that are derived from it and that some of these methods can be overridden. The virtual method **CEventDriven::ReleaseInstance** is one of these methods. **CEventDriven::ReleaseInstance** is called when there are no more events left to be processed by the **CServicingThread** for the object. This means that the object inheriting from **CEventDriven** can safely be removed from memory. By default, the **CEventDriven** implementation of this method calls MX_DELETE(this). However it might be required under certain circumstances to continue the destruction process asynchronously. For example, the object might depend on the asynchronous destruction of other packages. If this is required, the **CEventDriven::ReleaseInstance** method must be overloaded to delay its destruction. When the asynchronous destruction process is completed, the object can be deleted with MX_DELETE(this).

6.2.6 ECOM Release Example

This example shows an **ECOM** object that is also a **CEventDriven** object. This presents a problem because ECOM objects must be released by using the **IEComUnknown::ReleaseIfRef** method, but the **CEventDriven** object must be released with the **CEventDriven::Release** method. The solution to this problem is to override some **ECOM** release mechanisms to delegate the actual releasing to the **CEventDriven** class.

```
class CNetworkingService : public CEComDelegatingUnknown,
                           public CEventDriven)
{
    // ...

protected:

    // Override from CEComUnknown::UninitializeInstance. This method
    // will be called when the reference count of the ECOM object drops to 0.
    virtual void UninitializeInstance(OUT bool* pbDeleteThis)
    {
        // Call the base class.
        CEComDelegatingUnknown::UninitializeInstance(pbDeleteThis);

        // Don't let the CEComUnknown object proceed with the normal deletion.
        *pbDeleteThis = false;

        // Proceed with our own custom release which will eventually delete
        // the object after it is safe to do so.
        CEventDriven::Release();
    }
};
```

Code Sample 32 - CEventDriven::Release Example

7. Asynchronous Socket Factory

The Framework provides a centralized asynchronous socket factory that helps the creation of asynchronous sockets. The factory is extensible and allows an application to create its own specialized sockets by implementing the various **IAsyncSocketX** ECOM interfaces and registering a creation manager to create such sockets. Along with a creation manager, the application can also register a configuration manager that allows the application to set specific socket options on all (or only some of the) created sockets in the system. The socket factory also keeps track of all created sockets and allows the application to retrieve those sockets to perform operations on them.

These features are detailed in the following sections.

7.1 Creating a Basic Asynchronous Socket

The Framework supports UDP, TCP and TLS asynchronous sockets. To create a basic asynchronous socket, the **CAsyncSocketFactory::CreateAsyncSocket** method must be used. This method takes four parameters. The first is a pointer to the servicing thread on which the asynchronous socket is activated. If this parameter is NULL the socket will create its own servicing thread. The second is the type of the socket provided as an array of strings provided as a sequence of network protocols starting from the highest level down to the lowest level followed possibly by arguments. The third is the size of the previous array of strings. The fourth parameter is the location to where the new asynchronous socket will be returned.

The following code sample provides an example on how to create a basic asynchronous UDP socket.

```
// The location of the socket.
IAsyncSocket* pAsyncSocket;

// The type of the socket.
const char* apszUdpType[] = { "UDP" };

CAsyncSocketFactory::CreateAsyncSocket(NULL,
                                       apszUdpType,
                                       1,
                                       &pAsyncSocket);
```

Code Sample 33 – CAsyncSocketFactory::CreateAsyncSocket Example

The type of the socket to be created is determined by the array of strings passed to **CAsyncSocketFactory::CreateAsyncSocket**. Here are some examples of the possible array of strings:

```
      { "RTCP", "UDP" }, 2
      { "RTP", "UDP" }, 2
      { "RTP RTCP", "UDP" }, 2
      { "SIP", "TCP, m=client"}, 2
      { "SIP", "TCP, m=server" }, 2
      { "SIP", "TLS, m=client", "TCP" }, 3
      { "SIP", "TLS, m=client", "TCP, m=client" }, 3
      { "SIP", "TLS, m=server", "TCP" }, 3
      { "SIP", "TLS, m=server", "TCP, m=server" }, 3
      { "SIP", "UDP" }, 2
      { "STUN", "TCP, m=client" }, 2
      { "STUN", "TCP, m=server" }, 2
      { "STUN", "TLS, m=client", "TCP" }, 3
      { "STUN", "TLS, m=client", "TCP, m=client" }, 3
      { "STUN", "TLS, m=server", "TCP" }, 3
      { "STUN", "TLS, m=server", "TCP, m=server" }, 3
      { "STUN", "UDP" }, 2
```

Code Sample 34 – CAsyncSocketFactory::CreateAsyncSocket Array of Strings Example

The default socket types supported by the Framework are defined as constants in the files **Network/ MxAsyncSocketConstants.h** and **Tls/MxTlsSocketConstants.h** and are:

```
static const char* const gs pszUDP = "UDP";
static const char* const gs pszTCP = "TCP";
static const char* const gs pszOPTION CLIENT = "m=client";
static const char* const gs pszOPTION ACCEPTED = "m=accepted";
static const char* const gs pszOPTION SERVER = "m=server";

static const char* const gs apszSOCKET TYPE UDP[] = { gs pszUDP };
static const char* const gs apszSOCKET TYPE TCP CLIENT[] = { "TCP, m=client" };
static const char* const gs apszSOCKET TYPE TCP ACCEPTED[] = { "TCP, m=accepted" };
static const char* const gs apszSOCKET TYPE TCP SERVER[] = { "TCP, m=server" };
static const char* const gs pszTLS = "TLS";

static const char* const gs apszSOCKET TYPE TLS CLIENT[] = { "TLS, m=client", "TCP,
          m=client" };
static const char* const gs apszSOCKET TYPE TLS ACCEPTED[] = { "TLS, m=accepted", "TCP,
          m=accepted" };
static const char* const gs apszSOCKET TYPE TLS SERVER[] = { "TLS, m=server", "TCP,
          m=server" };
```

Code Sample 35 – Asynchronous Sockets Strings Constants

Asynchronous sockets report events through their event managers. The application must implement the managers and set them on the socket after it is created. For example, for an UDP socket, the managers that are needed are **IAsyncSocketMgr**, **IAsyncClientSocketMgr**, **IAsyncIoSocketMgr**, and **IAsyncUnconnectedIoSocketMgr**. The sockets report different events through these interfaces. For example, when a socket has data available to be received, **EvAsyncIoSocketMgrReadyToRecv** is reported on the manager implementing **IAsyncIoSocketMgr.** After receiving **EvAsyncIoSocketMgrReadyToRecv**, the application should call **Recv** on the asynchronous socket.

7.2 Custom Socket Types and the Creation Manager

The Asynchronous Socket Factory offers an interface that the application can implement to create custom socket types. This interface is the **IAsyncSocketFactoryCreationMgr** interface. Note that this interface is totally optional and, unless an application needs to define its own socket types, it can be left unimplemented.

A custom socket must be an ECOM object and implement some of the **IAsyncSocketX** interfaces. All custom sockets must implement the **IAsyncSocket** interface. Client sockets must implement the **IAsyncClientSocket** and **IAsyncIoSocket** interfaces. TLS client sockets must also implement the **IAsyncTlsSocket** interface. Server sockets must implement the **IAsyncServerSocket** interface. TLS server sockets must also implement the **IAsyncTlsServerSocket** interface. Sockets can also implement some of the following optional interfaces: **IAsyncSocketUdpOptions**, **IAsyncSocketTcpOptions**, **IAsyncSocketBufferSizeOptions**, **IAsyncSocketQualityOfServiceOptions**, **IAsyncSocketWindowsGqosOptions**.TLS sockets can also implement the **IAsyncSocketTlsRenegotiation** optional interface.

For example, to create the asynchronous custom socket type **CCustomAsyncSocket**, the application must first declare a CLSID for it. This can be done with a call to **MX_DECLARE_ECOM_CLSID(CCustomAsyncSocket)**. The application must register this CLSID with the ECOM factory by calling **RegisterECom(CLSID_ CCustomAsyncSocket, CCustomAsyncSocket:: CreateInstance)**. Before exiting, the application should un-register the CLSID using **UnregisterECom(CLSID_ CCustomAsyncSocket)**.

The declaration of the **CCustomAsyncSocket** custom socket could look as follows:

```
class CCustomAsyncSocket : protected CEComDelegatingUnknown,
                           protected IAsyncClientSocket,
                           protected IAsyncIoSocket,
                           protected IAsyncSocket
{
//-- Published interface
public:
    // The following statement is essential, it provides the default and unique
    // implementation of the IEComUnknown interface that every other interfaces
    // inherit from.
    MX DECLARE DELEGATING IECOMUNKNOWN

//-- Published methods through inherited interfaces
protected:
    static mxt result CreateInstance(IN IEComUnknown* pOuterIEComUnknown,
                                     OUT CEComUnknown** ppCEComUnknown);


    // Inherited from CEComDelegatingUnknown.

    virtual mxt result NonDelegatingQueryIf(IN mxt iid iidRequested,
                                            OUT void** ppInterface);

    // Inherited from IAsyncClientSocket.

    virtual mxt result BindA(IN const CSocketAddr* pLocalAddress);
    virtual mxt result ConnectA(IN const CSocketAddr* pPeerAddress);
    virtual mxt result SetAsyncClientSocketMgr(IN IAsyncClientSocketMgr* pMgr);

    // Inherited from IAsyncIoSocket.

    virtual mxt result GetPeerAddress(OUT CSocketAddr* pPeerAddress) const;
    virtual mxt result Recv(OUT CBlob* pData);
    virtual mxt result Recv(OUT uint8 t* puData,
                            IN unsigned int uCapacity,
                            OUT unsigned int* puSize);
    virtual mxt result Send(IN const CBlob* pData,
                            OUT unsigned int* puSizeSent);
    virtual mxt result Send(IN const uint8 t* puData,
                            IN unsigned int uSize,
                            OUT unsigned int* puSizeSent);
    virtual mxt result SetAsyncIoSocketMgr(IN IAsyncIoSocketMgr* pMgr);

    // Inherited from IAsyncSocket.

    virtual mxt result Activate(IN IEComUnknown* pIEComUnknown);
    virtual mxt result CloseA(IN ISocket::ECloseBehavior eCloseBehavior);
    virtual mxt result GetHandle(OUT mxt hSocket* phSocket) const;
    virtual mxt result GetLocalAddress(OUT CSocketAddr* pLocalAddress) const;
    virtual mxt result GetLocalInterfaceAddress(OUT CSocketAddr* pLocalInterfaceAddress)
          const;
    virtual mxt result GetOpaque(OUT mxt opaque* popq) const;
    virtual mxt result GetServicingThreadIEcomUnknown(OUT IEComUnknown** ppIEComUnknown)
          const;
    virtual mxt result GetSocketType(OUT ISocket::ESocketType* peSocketType) const;
    virtual mxt result SetAsyncSocketMgr(IN IAsyncSocketMgr* pMgr);
    virtual mxt result SetOpaque(IN mxt opaque opq);
    virtual const char* const* GetSocketType(OUT unsigned int* puSize) const;
    virtual mxt result SetSocketType(IN const char* const* ppszType,
                                     IN unsigned int uTypeSize);
    virtual mxt result EraseAllUserInfo();
    virtual mxt result EraseUserInfo(IN const char* pszUserInfo);
    virtual mxt result GetUserInfo(IN const char* pszUserInfo,
                                   OUT CBlob* pblob) const;
    virtual mxt_result InsertUserInfo(IN const char* pszUserInfo,
```

```
                                    IN const CBlob* pblob);

//-- Hidden Methods
private:

    // Inherited from CEComDelegatingUnknown.
    virtual unsigned int NonDelegatingReleaseIfRef();
};
```

Code Sample 36 – CCustomAsyncSocket Declaration Example

The ECOM methods implementation of the **CCustomAsyncSocket** should look like:

```
mxt result CCustomAsyncSocket::CreateInstance(IN IEComUnknown* pOuterIEComUnknown,
                                              OUT CEComUnknown** ppCEComUnknown)
{
    *ppCEComUnknown = MX NEW(CFilteringCustomSocket)(pOuterIEComUnknown);
    return resS OK;
}

unsigned int CCustomAsyncSocket::NonDelegatingReleaseIfRef()
{
    // Call base class for default behaviour.
    unsigned int uRefCount = CEComDelegatingUnknown::NonDelegatingReleaseIfRef();

    // If the socket has been created through the socket factory, there will
    // always be 1 reference on the socket for the socket factory's list
    // reference. It must be removed for the socket to be completely released.
    // That reference should always be the last one.
    if (uRefCount == 1)
    {
        mxt result res = resSI FALSE;

        // Remove the reference in the socket factory list.
        res = CAsyncSocketFactory::RemoveSocketFromFactoryList(this);

        // If the socket has not been created with the socket factory, it
        // will not be found in the socket list, and this means that the
        // application still has a real reference on the socket. Thus, the
        // real reference count must be returned.
        if (res == resSI TRUE)
        {
            //That was the last reference. Let the caller know this fact.
            uRefCount = 0;
        }
    }

    // It is extremely important that no operations on the socket's data
    // member or method calls are done at this point. This is because the
    // call to RemoveSocketFromFactoryList will possibly re-enter this method
    // and the reference count will fall to 0. This in turn will call
    // MX DELETE(this).
    //
    // So at this point in the method, it is possible that the 'this' pointer
    // is invalid.

    return uRefCount;
}

mxt result CCustomAsyncSocket::NonDelegatingQueryIf(IN mxt iid iidRequested,
                                                    OUT void** ppInterface)
{
    if (IsEqualEComIID(iidRequested, IID IAsyncClientSocket))
    {
        *ppInterface = static_cast<IAsyncClientSocket*>(this);
```

```
    }
    else if (IsEqualEComIID(iidRequested, IID IAsyncIoSocket))
    {
        *ppInterface = static cast<IAsyncIoSocket*>(this);
    }
    else if (IsEqualEComIID(iidRequested, IID IAsyncSocket))
    {
        *ppInterface = static cast<IAsyncSocket*>(this);
    }
    else
    {
        return CEComUnknown::NonDelegatingQueryIf(iidRequested, ppInterface);
    }

    static cast<IEComUnknown*>(*ppInterface)->AddIfRef();
    mxt result res = resS OK;
    return res;
}
```

Code Sample 37 – CCustomAsyncSocket ECOM Methods Implementation

Once the custom socket type is fully implemented, the application must implement a creation manager for it by implementing the **IAsyncSocketFactoryCreationMgr** interface. The application MUST register its implementation through **CAsyncSocketFactory::RegisterCreationMgr**. After this registration, a call to **CAsyncSocketFactory::CreateAsyncSocket** will call the **EvCreationRequested** event on the registered creation manager. The application should use **CAsyncSocketFactory:: UnregisterCreationMgr** before it exits.

The creation manager should look for a particular type of socket and create its own custom socket when a request to create a socket of the overridden type is made. For example, if the **CCustomAsyncSocket** custom type is to be used for all SIP UDP sockets, a possible implementation for its creation manager is:

```
mxt result CSocketFactoryCreationMgr::EvCreationRequested(IN IEComUnknown* pServicingThread,
                                                          IN const char* const* apszType,
                                                          IN unsigned int uTypeSize,
                                                          OUT IAsyncSocket** ppAsyncSocket)
{
    mxt result res = resSI FALSE;

    if (apszType == NULL || ppAsyncSocket == NULL || uTypeSize == 0)
    {
        res = resFE INVALID ARGUMENT;
    }

    if (MX RIS S(res))
    {
        // Start with an invalid CLSID.
        MX DECLARE ECOM CLSID(Invalid);
        mxt clsid clsid = CLSID Invalid;

        // Parse apszType from the end to find the socket type {"SIP" "UDP"}.
        if (uTypeSize >= 2 &&
            strncmp(apszType[uTypeSize - 2], "SIP", 3) == 0 &&
            strncmp(apszType[uTypeSize - 1], "UDP", 3) == 0)
        {
            clsid = CLSID  CCustomAsyncSocket;
        }

        // Create the socket if it is a CCustomAsyncSocket.
        if (IsEqualEComCLSID(clsid, CLSID Invalid))
        {
            // Report resSI FALSE to let other creation manager handle
            // creation.
            res = resSI FALSE;
        }
```

```
        else
        {
            res = CreateEComInstance(clsid, NULL, ppAsyncSocket);

            if (MX RIS S(res))
            {
                res = (*ppAsyncSocket)->Activate(pServicingThread);

                if (MX RIS S(res))
                {
                    // The socket was sucessfully created.
                    res = resSI TRUE;
                }
                else
                {
                    (*ppAsyncSocket)->ReleaseIfRef();
                    *ppAsyncSocket = NULL;
                }
            }
        }
    }

    return res;
}
```

Code Sample 38 – EvCreationRequested Implementation

7.3 The Configuration Manager

The Asynchronous Socket Factory offers an interface that the application can implement to configure sockets. This interface is the **IAsyncSocketFactoryConfigurationMgr** interface. Note that this interface is totally optional and that unless an application defines its own socket type or requires a very specific set of options on all its sockets, it can be left unimplemented.

Once implemented, the application MUST register its implementation through **CAsyncSocketFactory::RegisterConfigurationMgr**. The application should use **CAsyncSocketFactory::UnregisterConfigurationMgr** before it exits.

All configuration managers are called automatically for accepted sockets. However, for client sockets, the application MUST manually use the **CAsyncSocketFactory::CallConfigurationMgr** method with the client socket to configure as a parameter.

For example, to enable the TCP keep alive option on all TCP sockets, the implementation of **IAsyncSocketFactoryConfigurationMgr** should look like:

```
mxt_result CAsyncSocketFactoryConfigurationMgr::EvConfigurationRequested(
                                        IN const char* const* apszType,
                                        IN unsigned int uTypeSize,
                                        INOUT IAsyncSocket* pAsyncSocket)
{
    mxt_result res = resSI_FALSE;

    if (apszType == NULL || uTypeSize == 0 || pAsyncSocket == NULL)
    {
        res = resFE_INVALID_ARGUMENT;
    }

    if (MX_RIS_S(res) && uTypeSize > 0)
    {
        // Only configure TCP sockets.
        if (strncmp(apszType[uTypeSize - 1], "TCP", sizeof("TCP") - 1) == 0)
        {
            IAsyncSocketTcpOptions* pAsyncSocketTcpOptions = NULL;

            res = pAsyncSocket->QueryIf(&pAsyncSocketTcpOptions);

            if (MX_RIS_S(res))
            {
                // Enable the keep alive.
                pAsyncSocketTcpOptions->SetKeepAlive(true);
            }

            // Release ECom interfaces.
            if (pAsyncSocketTcpOptions != NULL)
            {
                pAsyncSocketTcpOptions->ReleaseIfRef();
                pAsyncSocketTcpOptions = NULL;
            }
        }
    }

    return res;
}
```

Code Sample 39 – EvConfigurationRequested Implementation

7.4 Retrieving Existing Sockets

The Asynchronous Socket Factory provides access to the existing sockets through the public and static method **CAsyncSocketFactory::GetSocketList**. That method returns a **CList** of **IAsyncSocket** pointer, the application can then perform operations on the existing sockets through this interface.

It is important to note that the **IAsyncSocket** interface is an ECOM interface and **CAsyncSocketFactory::GetSocketList** counts a reference for each of the pointers returned in the **CList**. This means that the application **MUST** call **ReleaseIfRef** on each pointer before it clears the **CList**.

The following is an example on how to fetch and use the socket list.

```
// Declare a CList of IAsyncSocket pointers
CList<IAsyncSocket*> lstSockets;

CAsyncSocketFactory::GetSocketList(OUT &lstSockets);

unsigned int uIndex = 0;
unsigned int uSize = lstSockets.GetSize();
for (; uIndex < uSize; uIndex++)
{
    // To send a packet on the socket, the IAsyncIoSocket interface is required.
    IAsyncIoSocket* pIoSocket = NULL;
    lstSockets[uIndex]->QueryIf(OUT &pIoSocket);

    // It can happen that the IAsyncIoSocket interface is not supported, this is the case
    // with unconnected UDP sockets and server sockets.
    if (pIoSocket != NULL)
    {
        // Send data…
        CBlob blob;
        unsigned int uSizeSent = 0;
        pIoSocket->Send(blob, OUT &uSizeSent);

        // Release the interface.
        pIoSocket->ReleaseIfRef();
        pIoSocket = NULL;
    }

    // Done with this socket, release it.
    lstSocket[uIndex]->ReleaseIfRef();
    lstSocket[uIndex] = NULL;
}

lstSocket.EraseAll();
```

Code Sample 40 – CAsyncSocketFactory::GetSocketList Example

8. Smart Pointers

8.1 Advantages of Using Smart Pointers

A smart pointer is an abstract data type that simulates a pointer. It adds the additional feature of a garbage collector. This is intended to reduce the risks of memory leaks by making sure that the resources associated with a pointer are always released when it is no longer needed.

The M5T Framework offers two types of smart pointer classes: CAutoPtr, which manages a pointer of any type, and CSharedPtr, which manages a pointer to a reference-counted object, such as an ECOM object.

8.2 CAutoPtr

The CAutoPtr class manages the lifetime of an object allocated dynamically by using the MX_NEW macro. When the CAutoPtr destructor is invoked, the object it manages is automatically released by using the MX_DELETE macro. This facilitates pointer operations since the user does not have to worry about releasing memory. It also reduces the risks of memory leaks.

In the following example, the memory allocated for the data structure is automatically released when the function exits.

```
void function()
{
    CAutoPtr<SDataStructure> spDataStructure(TO MX NEW(SDataStructure));

    spDataStructure->m_uValue = 42;
}
```

Code Sample 41 – CAutoPtr Example

8.3 CSharedPtr

The CSharedPtr class manages the lifetime of a reference counted object, such as an ECOM object. Please note that in order for an object to be eligible for management via the CSharedPtr class, it only needs to implement a reference count mechanism that uses the AddIfRef() and ReleaseIfRef() methods. The object needs to release itself once its reference count reaches 0.

When the CSharedPtr destructor is invoked, the reference to the object it manages is automatically released. This facilitates reference management since the user does not have to worry about calling ReleaseIfRef(). It also reduces the risks of memory leaks.

In the following example, the reference counted ECOM object is automatically released when the function exits.

```
void function()
{
    CSharedPtr<IInterface> spInterface;

    mxt result res = CreateEComInstance(CLSID CInterface, NULL, OUT spInterface);

    if (MX RIS S(res))
    {
        spInterface->InterfaceMethod();
    }
}
```

Code Sample 42 – CSharedPtr Example

8.4 Common Methods

Both the CAutoPtr and CSharedPtr have methods in common that behave the same way.

8.4.1 Get()

The Get() method of the CAutoPtr and CSharedPtr classes gives access to the pointer managed by these classes. This is particularly useful when passing a pointer to a method without giving the ownership of the pointer.

```
void function()
{
    CAutoPtr<SDataStructure> spDataStructure(MX_NEW(SDataStructure));

    spDataStructure->m_uValue = 42;

    DoSomething(spDataStructure.Get());
}
```

Code Sample 43 – Smart Pointer Get() Example

8.4.2 Release()

The Release() method of the CAutoPtr and CSharedPtr classes takes the ownership of the pointer managed by these classes. This is particularly useful when giving the ownership of the pointer to a method.

```
void function()
{
    CAutoPtr<SDataStructure> spDataStructure(MX_NEW(SDataStructure));

    spDataStructure->m_uValue = 42;

    // The method MUST call MX_DELETE on its parameter.
    DoSomething(TO spDataStructure.Release());
}
```

Code Sample 44 – Smart Pointer Release() Example

8.4.3 Reset()

The Reset() method of the CAutoPtr and CSharedPtr classes allows changing the value of the pointer managed by these classes. This is useful when the object being managed is no longer needed, and another object of the same type must be managed.

```
void function()
{
    CAutoPtr<SDataStructure> spDataStructure(MX_NEW(SDataStructure));
    SDataStructure* pNewDataStructure = MX_NEW(SDataStructure);

    spDataStructure->m_uValue = 42;

    spDataStructure.Reset(pNewDataStructure);
    pNewDataStructure = NULL;
}
```

Code Sample 45 – Smart Pointer Release() Example

9. References

9.1 M5T Coding Standard Reference

The M5T Coding Guidelines are available on the M5T Partners Web Portal (http://www.m5t.com/support-center.php) under the *Knowledge* tab.

9.2 Internet-Drafts and RFCs

- RFC 1035: DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION - http://rfc.net/rfc1035.html
- RFC 1769: Simple Network Time Protocol (SNTP) - http://rfc.net/rfc1769.html
- RFC 2308: Negative Caching of DNS Queries - http://rfc.net/rfc2308.html
- RFC 2409: The Internet Key Exchange (IKE) - http://rfc.net/rfc2409.html
- RFC 2460: Internet Protocol, Version 6 (IPv6) Specification - http://rfc.net/rfc2460.html
- RFC 2462: IPv6 Stateless Address Autoconfiguration - http://rfc.net/rfc2462.html
- RFC 2782: A DNS RR for specifying the location of services - http://rfc.net/rfc2782.html
- RFC 2915: The Naming Authority Pointer (NAPTR) DNS Resource Record - http://rfc.net/rfc2915.html
- RFC 3041: Extensions to IPv6 Address Autoconfiguration - http://rfc.net/rfc3041.html
- RFC 3174: US Secure Hash Algorithm 1 (SHA1) - http://rfc.net/rfc3174.html
- RFC 3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile - http://rfc.net/rfc3280.html
- RFC 3401: Dynamic Delegation Discovery System (DDDS) Part One: The Comprehensive DDDS -
- RFC 3402: Dynamic Delegation Discovery System (DDDS) Part Two: The Algorithm - http://rfc.net/rfc3402.html
- RFC 3403: Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database - http://rfc.net/rfc3403.html
- RFC 3513: IPv6 Addressing Architecture - http://rfc.net/rfc3513.html
- RFC 3526: More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE) - http://rfc.net/rfc3526.html
- RFC 3548: The Base16, Base32, and Base64 Data Encodings - http://rfc.net/rfc3548.html
- RFC 3596: DNS Extensions to Support IP Version 6 - http://rfc.net/rfc3596.html
- RFC 3761: The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM) - http://rfc.net/rfc3761.html
- RFC 3879: Deprecating Site Local Addresses - http://rfc.net/rfc3879.html
- RFC 4007: IPv6 Scoped Address Architecture - http://rfc.net/rfc4007.html
- RFC 4038: Application Aspects of IPv6 Transition - http://rfc.net/rfc4038.html
- RFC 4193: Unique Local IPv6 Unicast Addresses - http://rfc.net/rfc4193.html
- RFC 4634: US Secure Hash Algorithms (SHA and HMAC-SHA) - http://rfc.net/rfc4634.html

9.3 WWW References

Linux OS

- http://www.linux.com
- http://www.linux.org

Windows OS

- www.microsoft.com
- http://msdn.microsoft.com

VxWorks OS

- http://www.windriver.com/vxworks/

Nucleus OS

- http://www.mentor.com/products/embedded_software/nucleus_rtos/

Symbian OS

- http://www.symbian.com/
- http://www.newlc.com/
- http://wiki.forum.nokia.com/

Abstract Data Types

- http://en.wikipedia.org/wiki/Abstract_data_type
- http://www.zib.de/visual/people/mueller/Course/Tutorial/node4.html

Sockets

- http://en.wikipedia.org/wiki/Berkeley_sockets
- http://www.frostbytes.com/~jimf/papers/sockets/sockets.html
- http://www.sockets.com/winsock.htm

Open SSL Library

- http://www.openssl.org/

MIT Kerberos Library

- http://web.mit.edu/Kerberos/

Expat XML Library

- http://expat.sourceforge.net/

Regex Regular Expressions Library

- http://arglist.com/regex/

Index

Programmer's Guide

Media5 Corporation Inc.