Neoxen QX Framework

Developer's Guide

Table of Contents

Table of Contents 2
1 About This Guide 2
1.1 Audience 2
1.2 Organization 2
1.3 Typographic Conventions 3
1.4 Terms and Concepts 4
1.4.1 Abbreviations 4
1.4.2 Terminology 4
1.5 Related Documentation 5
2 Introduction 6
2.1 Introduction to Neoxen QX Framework 6
2.2 Purpose of This Guide 7
3 Background 8
3.1 Why Do We Need Guidelines and Standards? 8
3.2 How Neoxen QX Framework and this Guide Can Help 9
4 Designing Robust Software Solutions 11
4.1 Overview 11
4.2 Robustness 11
4.2.1 How to Plan and Verify Robustness? 12
4.2.2 Stress Tolerance 12
4.3 Adaptability 13
4.3.1 Estimating Required Adaptability 13
4.3.2 Keeping the Focus and Cost-efficiency 14
4.3.3 Adaptive Components and ROI 14
4.4 Security 15
4.5 Performance 16
4.5.1 Optimizing Network Traffic 16
4.5.2 Communication Modes 17
4.5.3 Load Balancing 18
4.5.4 Optimizing Connections to External Systems 19
4.5.5 Other Performance Aspects 20
4.6 Storing History Information 22
4.7 Changing the Foundation 23
4.7.1 Third Party Components 23
5 Creating a Professional Development Process 24
5.1 Development Environment 24
5.1.1 Microsoft® Windows® 24
5.1.2 Linux/UNIX 24
5.2 Software Builds 24
5.2.1 Configuration Management 25
5.2.2 Build Machine 25
5.2.3 Build Process 25
5.2.4 Build Tools 26
6 Source Code Tree Recommendations 28
6.1 Overview 28
6.2 Configuration Management 29

© Neoxen Systems

6.3 Workspace Settings in Visual Studio 2015 30
6.3.1 C/C++ Settings 31
6.3.2 Linker Settings 31
6.3.3 Resources Settings 31
7 General Formatting Guidelines 32
7.1 Source Code Readability 32
7.1.1 Tabs and Spaces 32
7.1.2 Positioning of Braces 33
7.1.3 Line Length in the Source Files 34
7.1.4 Size of a Logic Block, a Function and a File 34
7.2 Function Prototypes 34
7.3 Calling Functions with Multiple Parameters 35
7.4 Declaring Variables 36
8 General Coding Guidelines 37
8.1 Using Global Variables 37
8.1.1 Multithreading 37
8.1.2 Multiple Processors 37
8.1.3 Naming Convention 38
8.2 Using Static Variables 38
8.2.1 Multithreading and Multiple Processors 38
8.3 Initializing Variables 39
8.3.1 Using Constants 39
8.3.2 Reusing Constants 39
8.4 Using Strings 40
8.4.1 Standard C Library Functions 40
8.4.2 Localization Support 40
8.4.3 Pointer Arithmetic 40
8.5 Return Values 41
8.6 Error Handling 42
8.6.1 Validating Parameters 43
8.6.2 Validating Pointers 44
8.6.3 Structured Error Handling (SEH) 45
8.7 Using Macros 46
8.8 Memory Management 47
8.8.1 Desktop Applications 47
8.8.2 System Software 47
8.8.3 General Recommendations 48
8.8.4 Storage Allocation 48
9 Compiler and Linker Warnings 50
9.1 Using Explicit Typecasting 50
9.2 Using VOID Parameter with C Compiler 50
9.3 Using #pragma Directives 51
10 Using Resource Files 52
10.1 Resource Symbols 52
10.2 Version Information 52
10.2.1 Traditional Approach 52
10.2.2 Neoxen QX Framework Approach 52
11 Commenting Source Code 53
11.1 What to Comment? 53

© Neoxen Systems

11.1.1 File and Block Comments 53
11.1.2 Implementation Comments..... 54
12 Naming Convention 55
12.1 Hungarian Notation 55
12.1.1 Simple Data Types..... 55
12.1.2 Complex Data Types 56
12.1.3 Using Data Type Definitions..... 56
12.2 Naming String Pointers..... 57
12.3 Naming Functions and Methods 57
12.3.1 Using Module Prefixes 57
12.3.2 Using Logical Groups..... 58
12.4 Naming Constants..... 58
13 Exporting Functions 59
13.1 General Guidelines..... 59
13.2 How to Export?..... 59
13.2.1 Using Microsoft Specific Storage-class Modifiers 59
13.2.2 Using Module Definition Files 60
13.2.3 Using /EXPORT Specification 60
13.2.4 Calling Conventions for Exported Functions 61
13.3 Avoiding Multiple Header Inclusion 62
14 Solving Problems..... 63
14.1 Switching from Debug to Release Build 63
14.1.1 Heap Layout 64
14.1.2 Compilation 64
14.1.3 Code Optimization 65
14.1.4 Uninitialized Pointers 66
Index..... 67
APPENDIX I: Data Types 69
APPENDIX II: Data Type Prefixes 70

© Neoxen Systems

1 About This Guide

There are plenty of good guides and documents describing the design principles for creating sound and safe desktop applications.

Unfortunately, there are less good guides to describe the practical principles for designing well performing, robust and maintainable system components and distributed system applications.

This Developer's Guide describes the architectural and development guidelines, instructions, recommendations and usage information for the Neoxen QX Framework development platform.

This chapter provides basic information about this guide, such as the organization of the document, its intended audience and the typographic conventions and terminology used. It also lists other documents related to the topic.

1.1 Audience

This document is intended for software architects and developers utilizing Neoxen QX Framework. This guide assumes you are familiar with the basic concepts of client-server or multi-tier programming, C-style function calls, databases and networks.

1.2 Organization

This document is organized as follows

Chapter	Contents
Chapter 1	Describes the purpose of the document. It also explains the terminology and typographic conventions used in the document. A list of related documents can also be found in this chapter.
Chapter 2	Gives the introduction and overview to Neoxen QX Framework.
Chapter 3	Describes the background of Neoxen QX Framework.
Chapter 4	Describes the general guidelines for designing robust system software components and distributed business applications.
Chapter 5	Describes how to establish professional development processes.
Chapter 6	Describes the Neoxen QX Framework source code tree structure.

© Neoxen Systems

Chapter	Contents
Chapter 7	Describes the general recommendations for source code formatting.
Chapter 8	Describes the general programming guidelines.
Chapter 9	Describes the basic principles regarding compiler and linker warnings.
Chapter 10	Describes the usage of resource files.
Chapter 11	Describes the general source code commenting recommendations.
Chapter 12	Describes the recommended naming conventions.
Chapter 13	Describes the different approaches to exporting functions.
Chapter 14	Describes typical problems encountered when switching from debug to release mode.

1.3 Typographic Conventions

The following text styles identify special information used in this guide

Convention	Description
Italics	Italicized Text is used to call attention to cross-references.
`Courier`	Screen messages as well as literal user input, such as selections, commands, parameters and fields are written in `Courier` font.

Note: Important notes are written in this style.

Caution: Cautions are written in this style.

© Neoxen Systems

1.4 Terms and Concepts

The following abbreviations, terms and concepts are used in the document

1.4.1 Abbreviations

Abbreviation	Meaning, definition
CMS	Configuration Management System
CVS	Concurrent Versions System
HTTP	Hyper Text Transfer Protocol
JRE	Java Runtime Environment
LAN	Local Area Network
MFC	Microsoft Foundation Classes
ROI	Return of Investment
SEH	Structured Exception Handling
SQL	Structured Query Language
SVN	Subversion
TFS	Microsoft Team Foundation Server
WAN	Wide Area Network

1.4.2 Terminology

Term, Concept	Meaning, definition
Build Process	Fully documented process where the sources are stamped with build number, downloaded from CMS and typically a complete product is created with delivery media contents.
Build Engine	Fully documented and configurable software "engine" consisting of make-files, scripts and utilities to enable automated Build Process.
Build Machine	Computer with Clean Environment where the Build Engine produces the final product
Build Master	Nickname for the person in charge of the Build Engine
Clean Environment	Fully documented operating environment in a Build Machine containing only those components and software required for the Build Engine to execute.

© Neoxen Systems

1.5 Related Documentation

The following documents give you additional information

#	Document
[1]	Neoxen Modus Methodology - Development Guide
[2]	Neoxen Developer Resources Online

© Neoxen Systems

2 Introduction

2.1 Introduction to Neoxen QX Framework

Neoxen QX Framework is a 32/64-bit System Software Development Kit (SDK). It is the core of Neoxen QX technology and the foundation of all Neoxen products. It is a powerful SDK for distributed system software and multi-tier business and communications applications for multiple client and server operating systems. It scales from smart hand-held devices to clustered server environments.

Together with the included Visual Modus QX SDK it provides integration capabilities to any Information Management solutions supporting ODMA, WebDav or any of the standard TCP protocols. Additionally, it allows deep integration with Microsoft SharePoint 2010/2013/2016 and Bentley ProjectWise.

Neoxen QX Framework is designed for project groups, systems integrators and independent software vendors for rapid system software development. Also Neoxen Professional Services uses it as a foundation technology in all the customer projects requiring a robust, fast and efficient platform.

The origin of Neoxen QX Framework was in the evolving need to speed up development cycle of projects requiring robust communications interfaces. These design principles and visions have guided the development ever since. Globalization and an increasingly versatile infrastructure emphasize the need for a small, fast and highly flexible development platform even further.

© Neoxen Systems

2.2 Purpose of This Guide

The purpose of this guide is to illustrate the architectural software design guidelines and development practices recommended to be used with Neoxen QX Framework. The goal of this document is to define:

- General architectural design guidelines
- General development guidelines
- Guidelines for Neoxen QX Framework API usage

Some of the recommendations are not intended as strict rules. Common sense and a practical approach to everyday programming concepts can override these guidelines when necessary. However, it is strongly recommended that the exceptions should be well commented within the source code.

This Developer's Guide partially follows the Microsoft Windows SDK development recommendations together with some cross-platform requirements and it is targeted for C/C++ programming. However, the general guidelines are applicable to other languages with some modifications.

© Neoxen Systems

3 Background

3.1 Why Do We Need Guidelines and Standards?

When business applications are produced for customers, the possibility to offer a short and efficient project cycle with flexible and robust software results is a true winning combination. We need to standardize our processes, reduce redundant work and increase software re-usability and quality. We need to earn the competitive advantage and customer satisfaction by "doing the things right".

Architecture and design are the most important areas in successful software development. However, readability and maintainability of the source code are crucial parts in this goal as well.

The "hidden costs" in professional software development are some of the most important categories of expenses to minimize. This can be achieved only by creating, understanding and following standardized procedures and practices.

Unprofessional coding practices and implementation can ruin even the most intelligent design. Similarly, even the most systematic programming approaches cannot save poor design. In order to succeed in the professional software industry or in professional services we need standards and guidelines.

Creating and further developing professional procedures is a continuous task. Every software architect and engineer should participate in the process by following the common guidelines, which should cover at least:

- Solution guidelines and principles
- Development tools, their usage and recommended versions
- Configuration Management and Build Process
- Source code structure
- Coding practices
- Source formatting, commenting and documentation
- Declarations, calling and naming conventions
- Error handling and parameter checking

© Neoxen Systems

Project management, product management and quality assurance are the other important areas, which closely relate to development practices. Also customer care services, technical support and bug tracking are key elements. However, these areas are not covered in this Developer's Guide. They deserve their own guides.

3.2 How Neoxen QX Framework and this Guide Can Help

Neoxen QX Framework is highly flexible and customizable allowing as much control over the architectural layers as required. It permits developers and integrators to easily leverage the full power of Neoxen technology for their own benefit. By design, application development with Neoxen QX Framework allows for subsequent access to all the technologies and platforms supported by Microsoft and many others. The users of the platform are not forced to stick to a single solution or technology.

Neoxen technology specifically offers the following system development benefits:

- The modular architecture integrates easily with existing applications and services
- Neoxen technology works with many different commonly used technologies from multiple manufacturers
- Multi-layered architecture is extendable and customizable with all layers accessible to developers

Using Neoxen QX Framework as the main development platform also has a positive impact on development and maintenance costs reducing the ‘hidden costs’, whether it is implemented with a single application or within a network of different solutions. Following are the key areas where Neoxen QX Framework has a direct impact on overall project cost and effectiveness:

Save in Development Costs: Neoxen QX Framework is easy to learn and easy to use and the licensing is flexible and competitive. The resulting binaries are royalty-free. Neoxen QX Framework contains a consistent interface with lots of extensively tested modules. Therefore less coding is required to achieve the targets and the testing efforts can be concentrated on the business level functionality.

© Neoxen Systems

Build Better Solutions: As you save in development costs, you will also build better solutions utilizing the extensively tested and proven foundation. Less bugs, shorter testing cycles and shorter project completion will raise customer satisfaction and give a competitive advantage.

Standardize the Processes: Taking advantage of the guidelines drawn in this Developer's Guide you can refine and adjust your processes and increase efficiency.

Save in Support Costs: When there is a need to upgrade or modify your solution, the developers can typically concentrate on just the business level. Neoxen QX Framework is backwards compatible. If there is a need to upgrade the platform, the existing solutions remain fully functional and can even take advantage of some of the new features without re-coding.

One of the additional benefits of Neoxen QX Framework is that its usage is not restricted to any particular programming language or development environment. The core technology is written in low-level C language with C++ layer for optimal performance and inter-operability. Therefore, all the functionality is accessible from any programming language capable of calling C-style library functions (C, C++, C#, most scripting and macro languages, Java, etc.).

The whole code base of Neoxen QX Framework has been fully internationalized. Therefore the business applications created can be easily localized to any language area, including the Far East.

© Neoxen Systems

4 Designing Robust Software Solutions

4.1 Overview

There are some fundamental differences in the requirements of designing system components and multi-tier business solutions if compared to producing desktop applications.

Most of the system software requirements are common for all distributed systems regardless of their intended purpose. Technical requirements for servers and other system components are much more demanding than for desktop applications. Servers may typically need to stay up and running 24 hours a day, seven days a week, with quite challenging up-time requirements.

Desktop applications tend to have quite a short lifetime. Typically, they are used only certain limited amount of time before they are closed. Therefore, their design requirements are easier to manage and the design principles are easier to adopt.

It is quite common, that the system components need to be capable of managing very large amounts of data, reliably deal with heavy network traffic and efficiently handle multiple concurrent connections. These requirements set additional challenges for both design and implementation.

Neoxen QX Framework is designed from ground up to allow robust development of such software. This guide draws some basic guidelines to assist you in taking full advantage of the features and functionality implemented in this platform.

4.2 Robustness

Robustness and reliability should be common requirements for any software components. For system components this requirement has an additional emphasis. If a server component is not able to run reliably, it may cause unacceptable inconveniences to the using organization and increase the overall expenses.

© Neoxen Systems

4.2.1 How to Plan and Verify Robustness?

Software robustness is one of the most complicated areas to design and furthermore to verify. In a large software product, the number of possible combinations that should be taken into account and verify, may grow extensively. Without adequate testing and systematic approach, it is almost impossible to make any assumptions on a product's final robustness. This kind of planning and verification requires close communication between development groups and the testing teams.

If system software contains third party components, it may complicate proper design of reliability as well as testing. As far as robustness is concerned, the amount of unknown factors rises. These unknown factors may lead to a situation where not all the important combinations are fully understood. Also it may lead to a situation where test team may use more time than necessary to verify the 'combinations' that may actually use the very same internal workflow.

Some of the factors, which need to be taken into account, when planning for system software reliability:

- Supported platforms (Windows versions, Linux versions, other operating systems)
- Supported databases, their versions and drivers
- Supported communications features, protocol stack versions, etc.
- Supported Web Servers and/or Browsers and/or JREs if involved
- Possible 'adapters' and/or interfaces (for example Browsers / Web Services / native layers if supported, etc.)
- All other aspects that might cause software to behave differently than in the already tested combinations

4.2.2 Stress Tolerance

Estimating stress tolerance is one of the most critical design aspects. It is typically rather easy to get acceptable reliability results when the system component is running with quite a small load and perhaps with quite few concurrent users. Stress tolerance, capability of managing arbitrary load peaks and constant heavy workload is an aspect that has to be taken into account both in system design and in testing the software robustness.

© Neoxen Systems

There are quite a number of requirements to be considered, when planning the stress tolerance:

- Number of concurrent connections to other systems (database connections, stream socket connections, etc.)
- Number of concurrent client connections (or number of hits)
- Number of queued requests in each possible request queue
- Amount of data stored in memory at any one time
- Number of involved threads and processes
- Amount of other system resources in use
- Behavior under heavy system load caused by other applications

4.3 Adaptability

Globalization and an increasingly versatile infrastructure requires a quite a lot of adaptability from modern business applications and system software components. This kind of flexibility is needed for adapting the solution to varying environments with a minimum amount of effort and expense. The installation sites are different, customers have different workflows, network and server configurations and the number of concurrent users and the amount of expected workload may vary.

4.3.1 Estimating Required Adaptability

A good approach to estimating the required adaptability is to specify the minimum requirements the software should fulfill:

- What kind of networks and work practices should be supported?
- How large installations should be estimated?
- What platforms and databases should be supported in the beginning?
- Should there be an option to extend this support in the future?

Before starting any architectural design or implementation, it should be decided what are the basic requirements to be supported. Also there should be a view of at least some of the possible future needs. It might be wise to take into account some ‘requirements-to-come’ options during the design and implementation phases.

© Neoxen Systems

Should any of these or related requirements become important in the future, it might be relatively painless to extend the software to cover the arising needs without extensively changing the foundation architecture itself. Otherwise extending the software to cover unexpected requirements might be very tedious and costly. In the worst scenario this might lead to a partial or even complete redesign of the foundation architecture.

However, if unexpected functionality is requested for a reason or another, additional efforts will be needed in quality assurance. When more features are added or changed and the more they require changes in the foundation architecture, the more extensively the number of untested combinations grows.

In product development the main emphasis is typically on the next product release. If the possible future requirements are considered appropriately at the same time, the forthcoming release cycles might become much more economical to design, implement and test.

4.3.2 Keeping the Focus and Cost-efficiency

When adaptability is designed, it is always good to keep feet on the ground and use common sense. There is not a piece of software which can handle all the possible tasks and requirements efficiently. By keeping the business orientation in mind it is possible to be prepared to 'might-be-important-in-the-future' requirements with adaptive components without sacrificing the overall focus and cost-efficiency.

4.3.3 Adaptive Components and ROI

The adaptability design in the component level typically needs a more specific technical analysis:

- How components should be designed and implemented to best serve the minimal adaptability requirements?
- How easy it might be to leave the architecture more open and more configurable than required?
- Etc.

Many aspects should be considered, but one thing is quite obvious. In most cases it takes almost the same time to develop a generic solution than just a fixed design. An approach favoring open and adaptive solutions might save a lot of time and expenses in future development.

© Neoxen Systems

Adaptive components might be used in the future by the same system they were originally designed for, but they may also open new business opportunities and therefore increase the return of investment (ROI).

Shortly, adaptive components may address two different goals:

- To adapt the product to varying environment
- To decrease the design, development and testing costs of future projects

4.4 Security

Previously in closed local area networks (LAN) security considerations were not such an issue. Today when all systems should be in a wide area network (WAN), Internet compliant security is a cause of constant headache.

Even though some laws may set some of the requirements, most of the security considerations relate to confidentiality and privacy. It is important to identify what data is confidential or private and where all the software components might be located. We need to analyze which data transfers need security and by what means security should be provided.

Some years ago it was quite common that the original specifications came with an assumption that some system components were located in a closed and secure network area. Typically communication between those components was expected to be trusted and therefore fully secure.

Today these kinds of assumptions cannot be made. By default the component locations should be transparent so that the communication needs to be secured by the software designers.

© Neoxen Systems

4.5 Performance

4.5.1 Optimizing Network Traffic

If the performance of a business system is just fine in a local area network, it does not mean that the software still meets modern requirements. Companies have quite diverse networks these days and many organizations are geographically distributed. It is more and more common to work from home or use some temporary locations, such as hotels, airports etc. This sets many new requirements and expectations for the software.

Network traffic optimization is one of the 'new approaches'. For the younger generation of developers this may sound 'new'. For a seasoned programmer this may just be a return to good old days.

Optimizing network traffic may become quite critical when the system runs in a geographically distributed environment. Transporting data between two or more computers may from time to time be quite time consuming. Establishing a connection might take some time as well. It also may become important to ensure that a client does not lose the connection too frequently. The time needed to solve host addresses or route the packages, and network bandwidth in general, are all issues that have their impact in the performance and user experience. The performance might vary substantially in different environments.

There are lots of investigations and studies on what kind of responsiveness users find acceptable and what not. Respecting the user experiences and designing the software to meet those, may give strong competitive advantage. Network traffic optimization is one of the cornerstones in achieving this.

© Neoxen Systems

4.5.2 Communication Modes

There are two different basic concepts for connectivity, or rather communication modes. All distributed solutions use some sort of a variation of these

- Synchronous Communication Mode
- Asynchronous Communication Mode

Synchronous communication mode is quite often referred to as 'Connection Oriented Communication' and asynchronous mode is referred to as 'Task Oriented Communication'. Even though they are often used synonymously, that is not necessarily the case.

Synchronous Mode

Many traditional solutions relying on databases are synchronous by default. User connects to a database in the beginning of a session and the connection stays open until the application closes, or if a timeout value is reached without activity.

In practice this means that the client opens the connection to the server, executes desired operations while the connection is open and then closes the connection. In this case the connection has some kind of state maintained and it can be considered persistent.

Synchronous mode is typically used in local area networks. Connection persistence provides the best responsiveness and performance, but typically requires quite a lot of bandwidth.

Asynchronous Mode

Web applications and many messaging solutions are typically, but not exclusively, asynchronous. A typical example is e-mail. When e-mail is sent, the connection is opened for the execution period of the sending task and then closed. The sender does not have any certainty if the recipient ever received the message, unless the server politely informs so later on.

The used application level protocols are often stateless, such as HTTP in typical web applications. For the original purposes this is just perfect. However, it is more and more common to use asynchronous communication for traditional business purposes. In such cases there is a need to emulate connection persistence. Usually a persistent connection is needed to get the acceptable performance.

© Neoxen Systems

What methods can be used to get ‘persistent’ connections in asynchronous communication depends on the platforms that should be supported. One commonly used mechanism is to use cookies to store connection information. One possibility is to use features provided by other parties involved (like web server, web server version, etc.). This second approach differs a lot depending on what third party components are involved.

Architectural Requirements

From the architectural point of view these modes have different requirements for the software. Also the purpose of the software may dictate which mode to prefer.

However, it is not impossible to design the software to support both communication modes, they are not necessarily exclusive. If both aspects are properly taken into account in design and the architecture layered appropriately, software can enjoy the benefits of both modes with reasonable efforts. This approach may open completely new opportunities for the software and cause significant savings in future development.

4.5.3 Load Balancing

When large installations are concerned, horizontal scalability may have a significant role. It may become important to support multiple concurrent servers. Some servers may have identical roles and some servers may specialize in some specific tasks. This concept is typically referred to as Load Balancing. It might be that one host can run multiple server instances and also there might be multiple hosts involved.

Load Balancing can mean that there are multiple identical servers and one separate component takes care of delivering incoming requests to the servers in a way that the load on each server is feasible.

One typical approach is to deliver each incoming request to the server having the lowest load at that time. Another approach is server weighting according to their capacity. Some servers can efficiently manage heavier load than some other servers. Typically there is some kind of a configuration schema to specify the strengths or capabilities of the servers, or the balancing mechanism can even be fully automatic based on predefined algorithms.

© Neoxen Systems

Load balancing can be understood in various ways. For instance, manual load balancing might mean that there are no software components to handle balancing, but there are still multiple servers participating in the process. Alternatively, there might be some kind of configuration to specify which clients connect to which servers, or there might be hardware based load balancing, etc.

Furthermore, there might be multiple areas that may need load balancing. There might be multiple database connection needed for a session, or multiple connections to other systems. Some connections might be bound to a session and some operations might be handled by separate load balancing to a related target.

4.5.4 Optimizing Connections to External Systems

Very often server components need connections to external systems, such as databases. Design of these connections naturally affects performance. One basic guideline for database connections is to use as minimal a number of SQL statements as possible.
Architecture and general design of the connecting components may have a dramatic effect on overall performance. It can be seen too often that the modules involved in request processing are well designed from an object model point of view, but the design remarkably slows down system performance.

One of the most common examples is a situation where two or more methods in the same method chain separately execute the same SQL statement to access the same data. There is often a tremendous performance advantage if the component is designed so that the methods can use the same already fetched data.

Connection Pooling and Sharing

The principles mentioned above are usable for connecting with many kinds of external systems, such as systems that are accessed with stream sockets. Some design aspects affecting the performance should be considered as well. For instance, how the connections can be pooled, shared etc. and when the connection should be established and closed.

© Neoxen Systems

4.5.5 Other Performance Aspects

There are a lot of other performance related issues to consider. Quite many of them are easy to keep in mind and easy to achieve. For instance:

- To avoid executing operations, which are not necessarily needed
- To execute operations on locations where they most efficiently can be executed
- To avoid allocating memory and other resources before they are actually needed and releasing them in the right time

Some other aspects are not necessarily so obvious, but they can still have a strong impact on the system performance.

Designing Caches

Properly designed data cache is often something to consider:

- What kind of caches should be maintained and where?
- What kind of caching algorithms should be used?

In the traditional client/server solutions caches implemented in the client side typically increase the performance drastically. On the other hand, there is a lot of data that should not be cached. Improper caching schema might slow down the performance and might consume quite a lot of system resources. In many cases server side caches may even decrease the performance or prevent proper scalability.

However, there might be some critical data, often referred to, that may increase the performance if properly cached. In these scenarios intelligent caching algorithms may improve the performance significantly. Quite often there might also be some session state related data that should be cached in order to get a persistent connection.

© Neoxen Systems

Designing Initialization Routines

Reasonable amounts of consideration and sense should be used when designing data initialization and refreshing mechanisms. One principle is to avoid heavy initialization, loading or refreshing until it is absolutely necessary. In order to boost the performance, all the initializations should be as light as possible when a module gets loaded. It is not desirable to execute heavy initializations 'just in case'.

It might be that no other component refers to the loaded data or module. It is also possible that the data in question gets invalidated and needs to be reloaded before it is needed for the first time. This possibility also has an impact on refresh operations. If some kinds of caches are involved and a cache gets a refresh request, the data should only be invalidated and usually also freed. The reload itself should not happen until the data is referred to the next time (just-in-time reload).

Unnecessary updates may easily happen when object oriented programming is used improperly. This is especially common when objects have other objects as members. A typical mistake is to implement too much in the constructor. This might decrease performance as objects get created before there is a need to use any relating data, connection or other information.

Designing Memory Usage

It is quite important to design memory usage efficiently.

- How are large data blocks, containing large number of 'records' managed?
- How is memory actually allocated?

It should be clear without saying that the number of memory allocations and reallocations could drastically affect the performance. This is simple and easy to verify with timing tests.

For instance, there are typically operations that may return a large number of result records where the exact number is not known before allocating memory. A select statement is a good example: next records are received by subsequent requests until there is no more data available. It is not possible to define an optimal memory block size to support all cases. Some statements return only one record and some statements may return a million records.

© Neoxen Systems

One method quite often used is to always double the existing block size. In the end the block is reallocated to the exact size in order to prevent unnecessary memory usage.

With object oriented approach it is not that simple to use the most efficient solution. However, there are some questions that should be asked during the design:

- How many objects may get allocated and constructed implicitly, when additional objects should not be allocated at all?
- How many allocations may happen and how to maintain memory blocks so that they do not decrease the overall performance?

4.6 Storing History Information

In some business areas, such as insurance and telecom, there are official regulations on what kind of history information should be stored and for how long. Typically there is a need to store and archive security information, access logs, transaction logs, etc. Outside of the regulated areas these capabilities are too often neglected. Backing up and archiving database data is typically well taken care of, but in many cases this may not be adequate.

The product design should include a plan how the history information should be collected and managed:

- What kind of archiving the product should provide and what kind of support should be provided with utilizing third party implementations?
- What data should be archived at any one time and what parts of it could perhaps be archived independently from other data?
- Is the data that should be archived physically located in the same machine?

There are many ‘hybrid’ business solutions, which utilize both database and file systems. These kinds of systems cause additional challenges to software architects. Typically, some of the database data and related files need to be in synch and archived at the same time. Furthermore, the related files might be located in different machines.

4.7 Changing the Foundation

Proper foundation in the original architecture can protect from unnecessary redesigns. However, from time to time architectural changes may become necessary. Business directions may evolve differently than originally expected, or some other external reasons may require comprehensive modifications.

Changes of this scale typically have impact on the database schema and to the existing implementation. Changes may be required because of improving performance or to enable some new essential features. Some of the design and implementation aspects may require re-implementation or database conversion between different product versions.

New technology required in the implementation may require changes in the architecture. Also the implementation may need heavy rework, if the foundation architecture changes for one reason or another.

4.7.1 Third Party Components

A typical cause of changes is the third party components or engines used in the product. If that kind of module is replaced with another third party component or will be partially replaced with own implementation, rework cannot be avoided.

It is not uncommon, that third party components set some restrictions and rules for the product design. These restrictions may have a drastic impact on product performance or scalability, or they may hinder implementing new features. Sometimes it is realized that the reason to change the existing architecture is in the architecture itself. Maybe the third party component was not an appropriate selection for the product in the first place.

When planning to replace a third party component the existing implementation needs to be thoroughly analyzed. It is necessary to clarify how much of the implementation would need redesigning in order to support the new environment.

Usually this aspect is already taken into account when selecting the new third party component:

- However, was the same mistake made again?
- Was it properly investigated?
- Were the possible limitations and future problems the new component would bring with it verified?

© Neoxen Systems

5 Creating a Professional Development Process

5.1 Development Environment

It is highly recommended that all the developers attending the same project use the same development tools and same versions with the same service levels. The tools used should be decided at the beginning of the project and they should not be changed during the project cycle.

5.1.1 Microsoft® Windows®

We recommend using Microsoft Visual Studio® in C/C++/C# development. Currently version 2013 with the latest Service Pack and version 2015 are certified for usage with Neoxen QX Framework and therefore recommended. All coding and source code editing should be done in this environment. Within a development team it is not encouraged to upgrade to another version without prior agreement.

The editor in Visual Studio has its own indentation schema, which may be different from other editors. To keep the readability of all the sources at the optimal level, all the programmers are advised not to modify the sources with any other editing tool.

5.1.2 Linux/UNIX

The core components of Neoxen QX Framework support 32/64-bit Linux and many flavors of 32/64-bit UNIX. However, the commercial version is officially certified for Microsoft Windows platforms only.

5.2 Software Builds

Term 'Build' is used to describe the physical process of creating a software product. The process contains the Configuration Management System (CMS), a dedicated computer for creating the software product and separate command line tools and utilities for the actual build process.

For a professional 'Build' it is characteristic that the source code gets marked consistently with logically incremental version numbers, the process is as automated and unattended as possible and it can be repeated any time afterwards.

© Neoxen Systems

5.2.1 Configuration Management

Also for these reasons CMS, often also referred to as Version Control System, is an elementary component of a professional build system. Naturally there are many other important reasons for using a CMS system as well, such as concurrent development, conflict management, quality control, concurrent branches, change history, etc. For further details, please refer to the documentation of your chosen CMS.

Old wisdom says that it does not matter which CMS you use, as long as you use one. This is really true in a sense, that any CMS is better than no CMS at all. However, there are some aspects that might influence your decision on choosing your CMS platform. If you develop only for Windows, Microsoft Visual Source Safe may be a sound and solid option. However, if you believe you might need better support for other operating systems, such as Linux, some other alternatives might be more suitable. There are many commercial and open source alternatives available.

Neoxen QX Framework can be used with any of the widely used Configuration Management Systems. CVS and SVN are perhaps the most widely used Configuration Management System in the world. For instance, majority of the open source development, such as Linux, Apache, KDE, Gnome, etc. is done with CVS or SVN. Another important benefit is that it runs in almost any viable operating system. However, any of the commercial systems, such Microsoft Team Foundation Server is a good choice.

5.2.2 Build Machine

For creating the software product it is highly recommended to have a dedicated computer, often referred to as the ‘Build Machine’.

The build machine should have a fully documented operating environment containing only the selected software and components required for building the product. This is often referred to as a ‘Clean Environment’.

5.2.3 Build Process

The actual Build process is often referred to as a ‘Build Engine’. In this sense it typically means the source code tree with the relevant make-file hierarchy. There should be a dedicated person who is in charge of the build process. The person with this special responsibility is sometimes referred to as the ‘Build Master’.

© Neoxen Systems

The product sources are stamped with the version number and they are checked out from the CMS into the cleaned build machine. It is recommended, that the version numbering should contain at least three double digits (major version, minor version, build number), such as 10.56.12. At least the build number should always be incremented in every build.

A complete product build should result in the delivery of media contents. This means, that all the components for the product should be built within the same build process, including documentation, readme files, everything. Typically, this means that the automated build process produces the product CD/DVD contents as the final output.

5.2.4 Build Tools

Visual Studio conveniently generates make files to be used with its own nmake-utility. This is nice and simple, but not necessarily useful for a professional software build process, especially in multiplatform environments. There are many good reasons to separate the development environment and official build utilities. In small projects this may not be an issue, but when the project grows and the amount of modules increases, the value of the separation is imminent.

In Visual Studio the native make files are part of the living code base. In small projects this is an advantage, as they do not need manual maintenance. In larger projects and product development, this advantage turns into a burden and a constant issue for quality control. Every build should be repeatable at any time exactly as it was during the original build. Also the Build Engine should remain subject to intentional changes only. When the product has a larger user base and there are customers with different released versions, importance of these details may turn out to be crucial. This is especially important, if there are long-term customer commitments, maintenance agreements, etc.

Neoxen QX Framework provides a superior solution for this shortcoming. The product takes advantage of Gnu’s make-utility (qxgmake.exe) with some modifications made by Neoxen Systems. There are plenty of benefits in using external tools for the Build Process, on top of those mentioned above. In today’s fast evolving business environment support for other operating systems, including Linux, is something you should not ignore.

© Neoxen Systems

A professional Build Process should:

- Support multiple operating systems without any major modifications
- Prevent unintentional changes to make files
- Support scripting, environment variables, substituting and other control mechanisms

Neoxen QX Framework provides all this together with ready-made templates for building a complete, professional and high quality multi-platform Build Engine.

© Neoxen Systems

6 Source Code Tree Recommendations

6.1 Overview

Whenever a development project contains more than a few source files, the project should be built around a logical file system tree.

Traditionally for multi-platform compatibility reasons it is recommended, that all the files in the project should respect the 8+3 notation and all the names should be in lower case.

Visual Studio project workspace for each module should be created to the module's project directory. All references to directories located in the source tree should be defined using relative paths.

Output directories for object files, binary files, library files and all other files generated during compilation and linking process should be created to a pre-agreed location. Output files should never be compiled into any part of the source tree hierarchy. Often the output directory and the source directory are located in different drives, especially when producing release builds.

Subcontractors or other partners participating in the project who do not have direct access to the version control system should still map their tree structure according to the agreed convention. The sources are checked regularly and building a new project workspace or adjusting it each time is waste of time and money.

When building modules outside the Visual Studio using make files the output directory should be specified with an environment variable. In that case output directories can be freely chosen. Official release builds are always built using make files, scripts and command line tools in an automated process.

The directory structure of Neoxen QX Framework installation is described in Diagram 1. The source tree subdirectory structure for all the modules is documented in Diagram 2. The output directory structure is documented in Diagram 3.

© Neoxen Systems

6.2 Configuration Management

SVN (Subversion), typically maintained in Windows 2012 Server R2, Windows 2016 Server or Linux, is used as a source code version management example in this manual. The SVN-specific version control files are located in SVN directories in every workstation. These directories and their contents are automatically maintained by the version control system. These files must be left untouched and they should not be deleted.

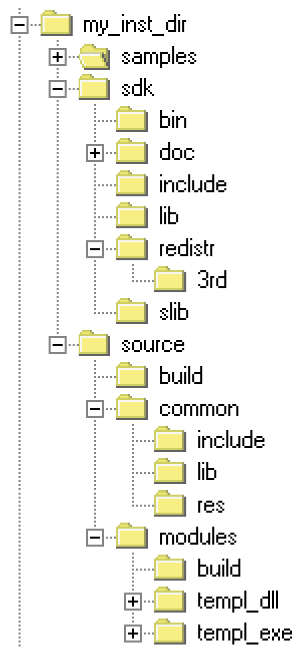

Figure 1
This diagram illustrates the high level hierarchy of the Neoxen QX Framework installation directory and the structure of the recommended source code tree. The module specific hierarchy is described in Diagram 2.

© Neoxen Systems

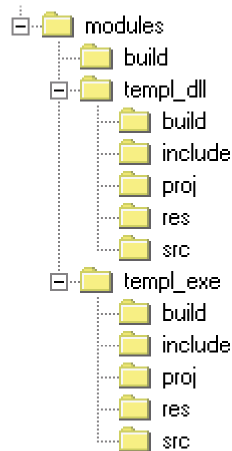

Figure 2
This diagram illustrates the recommended source code structure for each module. The Build directories contain ready-made make-file skeletons for your convenience.

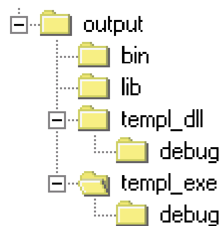

Figure 3
This diagram illustrates the output directory structure for Visual Studio project compilation.

6.3 Workspace Settings in Visual Studio 2015

When a new development project is established, the process starts by creating the standard directory structure and the Project Workspace in Visual Studio.

If the project is about to use MFC, static linkage is not recommended. This setting is defined in the '**<Project> Property Pages-Configuration Properties-General**' page. The paths for the intermediate files and for the resulting binaries are defined in the same page.

NOTE:	Always use relative instead of absolute paths. All the paths should be relative to the directory which contains the project workspace files. Use the Visual Studio internal variables as much as possible.

6.3.1 C/C++ Settings

The 'General' page contains the item '**Additional Include Directories**'. The correct paths for project specific header files should be added here.

If the project is using MFC, the following addition should be used. In the '**Preprocessor**' page in the 'Preprocessor Definitions' you should add _USRDLL definition.

6.3.2 Linker Settings

If the project contains any of those Windows import libraries, which are not in the default list, they should be added through the '**Input**' page. This page should be used only for compiler-specific import libraries and Neoxen QX SDK libraries.

6.3.3 Resources Settings

If the project contains custom resources or hand written headers for the resources used, the additional paths should be defined through the '**General**' page in the 'Additional Include Directories' setting.

© Neoxen Systems

7 General Formatting Guidelines

7.1 Source Code Readability

Code readability and formatting in general are much more important than someone might expect. In order to save time in code reviews and to help other developers getting acquainted with the code with less effort, the code should be ‘airy’ and easy to read. Tight writing is not same as tight coding.

Comments inside the code are required only to describe the logic of the code or to describe a ‘catch’ or special usage of some standard elements. Commenting as well as naming convention is discussed more in detail later in this guide.

7.1.1 Tabs and Spaces

‘Hard tabs’ should not be used in any event. Spaces should be used instead. By default, Visual Studio presents hard tabs with 3 spaces. However, this is a configurable option. In other environments this varies depending on the platform and the editor used.

With Neoxen QX Framework hard tabs should be expanded to 3 spaces. This setting must be set in Visual Studio via ‘**Tools-Options**’ menu at the ‘**All Languages-Tabs**’ page. Also ‘**Insert spaces**’ option should be selected in order to avoid hard tabs.

© Neoxen Systems

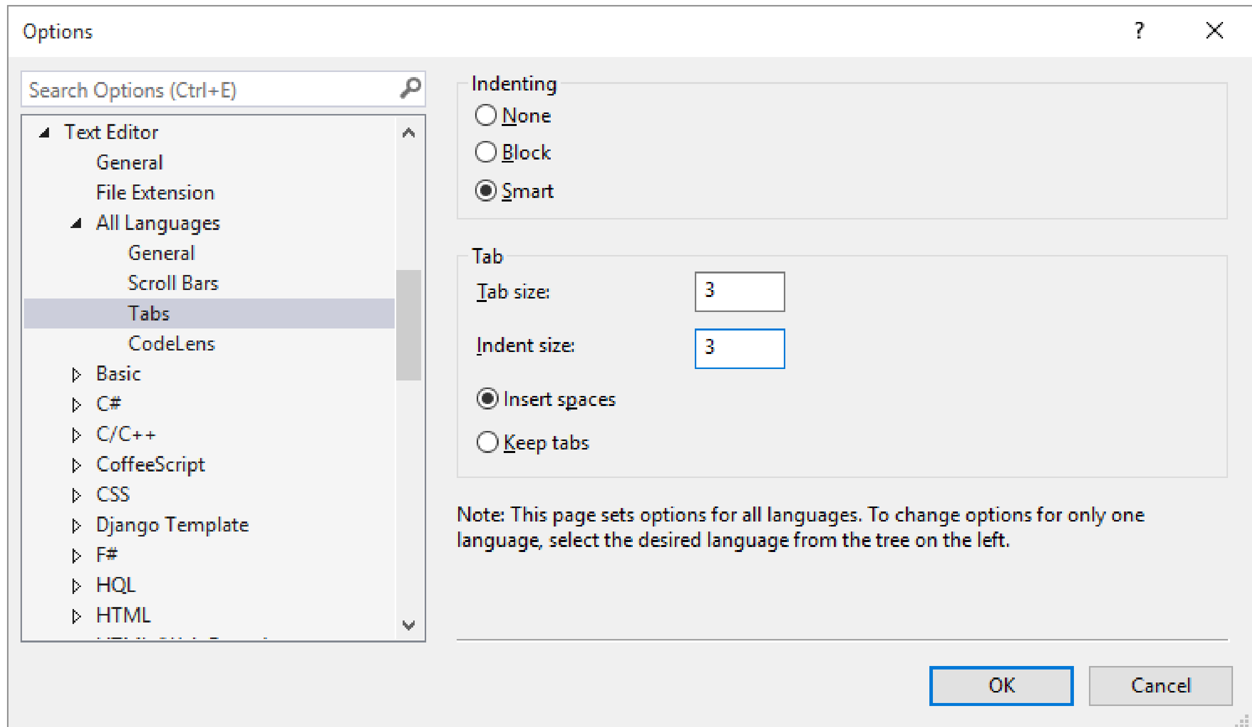

Figure 5
This screenshot displays the Visual Studio 2015 Text Editor All Languages Tabs Settings.

7.1.2 Positioning of Braces

How should the block braces be used? This is a question with two definite answers, depending on whom you ask it. However, with Neoxen QX Framework the left starting brace of a block is always recommended to be positioned on its own line.

```
for (iLoopIndex = 0; iLoopIndex < QXGGS_LOOPMAX; iLoopIndex++)
{
     if (QXGGS_DOIT == iLoopIndex)
     {
          QxDoWhatSoEver();
          continue;
     }
}         // End for
```

The position of the ending brace must correspond to the starting brace.

© Neoxen Systems

Note: In if- and else-statements and in loops, the block braces should always be used, even if it contains only one instruction. This clears the code and eases future updates and modifications.

7.1.3 Line Length in the Source Files

Line length in source code files should not exceed 80 characters. This helps to keep the code readable in different screen resolutions.

7.1.4 Size of a Logic Block, a Function and a File

Naturally there are no strict rules for these issues, just some general common sense recommendations.

If a logical block (if-else, loop, etc.) does not fit into a visible area in the Visual Studio editor, it might be wise to rethink the logic. Switch-case statements are typical exceptions.

The same guideline may, in some respect, be valid for functions and methods as well.

File sizes should be kept reasonable. Thousands of lines of code in a single source code file should raise some doubts on modularity.

7.2 Function Prototypes

Following example illustrates the format of a function declaration:

```
DWORD QXAPI QxMyOwnGetString
   (
   LPTSTR lptstrReturn, // o - Address of the buffer for a string
   DWORD  dwBuffSize    // i - Size of the buffer
   );
```

Function prototypes should not be introduced ‘on-the-fly’ or in the beginning of a source code file. In order to make the source code level quality control easier, all the prototypes should be placed in header files. Prototype formatting is recommended to be according to following principles:

- Function parameters are placed in the left side of the source code page

- Starting three spaces from the margin
- Each parameter is placed on its own line
- Each parameter is followed by a comment
- The comments using C++ -style commenting start with:
- i (input)
- (output)
- io (both input and output)

The input-output marker should be followed by a short description of the variable.

Generally the parameters should be in a following logical order:

- Input parameters
- Output parameters

Note: However, if the output parameter requires an additional parameter describing the storage allocation for it, it can appear after the output parameter. Look at the sample above.

7.3 Calling Functions with Multiple Parameters

When a function with multiple parameters is called, the parameters should all be placed in separate lines and vertically aligned. Short parameter lists can be either on a single line or each in their own line. 'Mixed mode' cannot be accepted.

```
MessageBox(hwnd,
           lptstrMsg,
           lptstrTitle, // Parameter usage can be commented
           MB_OK);
```

The formatting style used in the above sample is very useful for documenting special usage of some variables and highly recommended for all lower level functions.

© Neoxen Systems

7.4 Declaring Variables

For the sake of readability and maintainability all variables should be declared on separate lines. Also the documentation issues may insist this convention:

```
LPTSTR lptstrNetBuf = NULL;    // Buffer returned by API
LPTSTR lptstrParams = NULL;    // Buffer for the dynamic parameters
```

Note: All the pointers should be initialized when they are introduced.

In C++ code it is not a good habit to introduce variables on the fly. This approach easily leads to unnecessary variables, which may turn to unused variables as time goes by. Good housekeeping and discipline pays back in the long term. All the local variables should be introduced in the beginning of function or method. Local block-scope variables are the only exceptions.

© Neoxen Systems

8 General Coding Guidelines

8.1 Using Global Variables

Global variables always form a risk and a concern. Multithreading and multiple processors only work well with global variables if their usage is properly planned. As the old wisdom says, usage of global variables should be minimized and parameter passing should be favored instead. Perhaps global variables cannot be totally avoided, but when used, the scalability and consistency aspects must be well designed in advance.

The overall schema for using global variables should already be decided during the software design stage:

- What kind of data should or could be kept in project scope global variables and which is to be placed in file scope global variables?

The main principle is that project scope global variables should be completely avoided and file scope global variables used instead, if necessary.

8.1.1 Multithreading

Accessing global data most often requires thread locking to enable usage in a multithreaded application as well. Accessing the global data only in the scope of a single file also encapsulates lock management for the data into a single file. This approach helps to avoid the most typical synchronization bugs.

8.1.2 Multiple Processors

Also for scalability reasons, global data cannot be recommended. This is especially true in multi-processor/multi-core computers. Operating systems multitask threads, not processes. In a multiprocessor computer this means that different threads of the same process can be run in different processors. Accessing global data from different threads running in different processors always causes an unnecessary context switching. What this means is, that a badly designed application using global variables can lose some of its performance in a multiprocessor computer.

© Neoxen Systems

8.1.3 Naming Convention

However, if global variables need to be used, special consideration should be paid for their naming convention. Using the same name for a project scope global and file scope global is strictly forbidden. All the names of global variables should be prefixed in consistent and unique manner. It is recommended to use **g_** as a prefix for project scope global variables and **g** in file scope.

All the **project scope global** variables must be introduced with **extern** modifier in a single common header file. Introducing is done without initialization of the variables. Later on when taken into usage, all global variables of this type should be declared for usage, always with initial values, in the beginning of a single source code file. This file typically contains the project entry point.

Note: Variables which are designed to remain file scoped should never use the extern modifier.

8.2 Using Static Variables

As with the global variables explained earlier, special notice has to be put on static variables as well. They should be avoided in the same manner. Especially dynamic link libraries and static variables can be a very error prone combination. This is because of the way static variables are stored in memory.

When a shared library is loaded and a static variable internal to a function receives a new value, the value remains valid as long as the object code (single source file) is in the memory. This happens even if the function containing the static variable is exited. In practice this can cause several strangely appearing error conditions.

8.2.1 Multithreading and Multiple Processors

Without locking mechanisms static variables are not thread safe. In a manner similar to the global variables, static variables do not scale properly in multiprocessor computers.

© Neoxen Systems

8.3 Initializing Variables

We should always use initialized variables. Sometimes it may look unnecessary, but we should always remember the fact that code 'lives'. For the sake of future maintenance, it is a highly recommended way to protect our code from future bugs. Implementation of initial values is discussed in the context of constant values.

8.3.1 Using Constants

All constant values and literals should be defined in an appropriate header file. 'Hard coding' should not be accepted.

8.3.2 Reusing Constants

In order to keep the program logic under control and to decrease the expenses in maintenance, reusing constant definitions for different logical purposes should not be accepted.

Sample:

```
#define AXG_MAXLEN                    256
#define AXG_MAX_ITEMS                 256
#define AXG_MAX_SHOE_SIZE             256
```

In the sample above, the actual values of the definitions happen to be exactly the same, but they are logically used to completely different purposes.

If there is a definition, let us say for 256, in some of the header files, that particular definition should be used for one logical purpose only.

© Neoxen Systems

8.4 Using Strings

8.4.1 Standard C Library Functions

Copying and concatenating strings is a typical source of 'hidden' bugs. Especially **strcpy()** and **strcat()** are infamous examples of standard library functions, which should be avoided as carefully as possible. These kinds of functions should not be used, if the allocated size of the strings is not explicitly known.

Furthermore, if a function receives parameters of any string related data type, none of the above mentioned functions should be used without explicit pointer validation.

In case of concatenation, special care should be taken that the combined actual length of the strings does not exceed the storage allocation of the target buffer.

Visual Studio 2005 implemented new safer buffer manipulation functions, which should be used.

8.4.2 Localization Support

Generally, none of the platform specific or ANSI C string functions should be used directly. Large portions of those functions support only single byte character sets and are therefore not suitable for proper localization. The Neoxen QX Framework provides string manipulation functions and macros, which are cross-platform compatible and also support multi byte character sets.

8.4.3 Pointer Arithmetic

In the old times of DOS and Unix programming in plain C, pointer arithmetic was a widely accepted and encouraged method to achieve short and efficient code.

However, the requirements of modern 32/64-bit programming are different. Minimizing the 'hidden' costs of the development cycle and especially maintenance costs are one of the top priorities, especially in the application level. Also, overall code stability is one of the essential requirements.

© Neoxen Systems

Especially with strings the pointer incrementing and decrementing in the traditional way is not a proper way to code. We have to remember that in multi byte character sets one character can be one or two bytes in size. In Unicode each character takes two bytes.

In general, excessive use of pointer arithmetic is not encouraged. The only exception is in the low level code, where performance requirements are crucial.

Caution: Programmers should keep in mind, that when manipulating data received as a pointer, they are manipulating the original data. This may cause undesired side effects, if not understood correctly.

8.5 Return Values

When designing a new component, special notice has to be put on the return values. Consistent and unified return value schema has to be designed and implemented. Design of the return value implementation must be carefully thought over before a single line of code is written.

Utility functions should always return a success indicator as:

- Long integer
- Boolean
- Handle

As a general guideline, if there are no real reasons to return 16-bit integer values, 32-bit data types are recommended.

If a function is supposed to return a string or other pointers, it should receive a proper storage pointer as an output parameter. Returning a string pointer as a return value is not recommended, as it often leads to “buffer overflow” type of bugs.

Specifically, when a function receives a pointer to a buffer as an output parameter, it must be accompanied with a parameter specifying the size of allocated storage.

© Neoxen Systems

```
DWORD QXAPI QxMyGetString
(
   LPTSTR lptstrReturn,    // o - Returns a pointer to a string

   DWORD  dwBuffSize       // i - Allocated size for the buffer
);
```

When using integral return values, value defined as 0 is regarded as an indicator of successful operation.

Return values should always be defined in appropriate header files with descriptively defined names. This is discussed in more detail later in the context of naming conventions.

Applications that use Neoxen QX Framework API calls or their internal utility functions should explicitly handle the return values. A switch-case structure is the preferred method.

8.6 Error Handling

Error management starts from the design of the return value schema, as described previously. The main principle should be that if an API library function terminates with error, it should set the error code and message using error management functions published in QXERR.H. In most cases the failing function also returns the error code. The last error in the thread is only set if the function fails.

The last error can be set in two different ways. The first way is to set the error and overwrite the previous last error code and message. Another way is to extend last error. Extending the last error makes sense when the reason for the unsuccessful operation is the failure of another module function call and an error message is already set.

The application using the Neoxen QX Framework API calls is responsible for interpreting the codes.

© Neoxen Systems

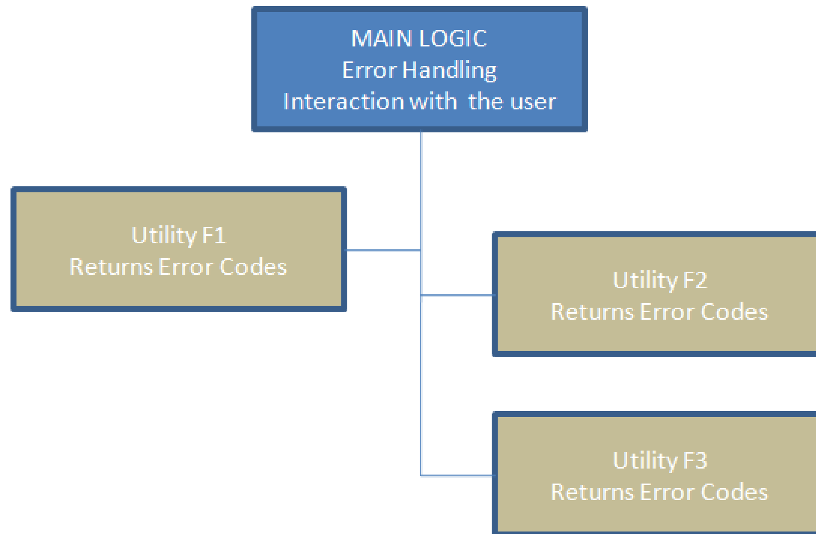

Figure 6
Special notice has to be put on the program logic. Error handling should be done in the highest possible level in the logic hierarchy. Utility functions should be designed to be as generic as possible and the responsibility of interpreting their return values should be taken care of by the high level functions.

8.6.1 Validating Parameters

All the mandatory parameters should always be validated. If a mandatory parameter is not valid, the code should not continue.

Logical Checking Order

Checking may not necessarily be done in the same exact order in which the parameters exist, but rather according to their logical order. Output parameters should be checked first and initialized if required. Then the rest of the parameters are checked, usually in the order they appear. The input parameters requiring more logic in checking, or which are validated by a separate function call, are checked last.

Pessimistic Checking Order

In many cases it is far more efficient to test for failure rather than success when checking return values of function calls. It also shortens the code and increases code readability.

Testing the success of a call leads easily to multiple levels of if-statements and makes the logic of the code more error prone, harder to read and more time consuming to maintain.

© Neoxen Systems

Instead, the programmers should consider each call in a 'pessimistic' way:

WHY CONTINUE, IF THE CALL FAILED?

This attitude often clarifies the program logic thus making it more robust against bugs. It saves a lot of effort in maintenance and therefore may save considerably in the forthcoming development expenses.

Note: Common sense must be used in the checking order. In some cases there may be a situation, where every exit point requires multiple cleanup actions. In this kind of situation, optimistic checking order may produce better code.

Note: In general, 'goto'-statements should not be used.

8.6.2 Validating Pointers

If a function receives pointers as parameters, they should always be validated carefully. The only exception is when a NULL pointer is specifically allowed.

Neglecting pointer validation leads sooner or later to undetermined instability of the software therefore raising the overall expenses of the development and especially maintenance.

© Neoxen Systems

```
DWORD QXAPI QxMyFunction
(
   LPTSTR lptstrReturn,        // o     Returns the name…
   DWORD  dwBuffSize           // i     Size of the return buffer
)
{
   if (!lptstrReturn)
   {
      return QXGGC_ERR_INVALID_PARAM;
   }

   if (0 <= dwBuffSize)´
   {
      return QXGGC_ERR_INVALID_PARAM_SIZE;
   }

…
```

8.6.3 Structured Error Handling (SEH)

All functions should take care of their parameter validation, as described. The try-catch blocks are used when the execution can cause an exception. For example, all callback functions should be executed within a try block.

```
QXAPI_TRY
{
    // statements that may cause an exception
}
QXAPI_CATCH
{
    // this catch block will handle all C++ and C style exceptions
    // the error code is set here and the value indicating the
    // unsuccessful operation is returned
}
```

© Neoxen Systems

All modules should be implemented so that stability is one of the fundamental requirements. One thing to remember is that a NULL check does not validate a handle or a pointer. Appropriate SEH blocks should be used when objects are referred using handles or pointers that are passed as parameters for exported functions. All statements that might cause exception (like _stprintf when the format string is not fully validated or _tcscpy when the destination buffer is received as an input parameter) should also be executed using a SEH block.

If a function causes memory overwriting, then SEH block does not necessarily protect the running application. The severity of the possible damages depends on the destroyed contents of the overwritten memory area.

The SEH block implementation in Neoxen QX Framework expands to try – catch statements only in release builds. On debug builds the SEH Blocks always expand to if (1) - else statements. This prevents unintentional "protection" of code blocks in debug builds so that possible bugs are easier to locate.

Note: Please do not use platform or compiler specific try-catch mechanisms. Neoxen QX Framework implements a consistent Structured Error Handling mechanism, which supports both C and C++.

8.7 Using Macros

The major difference between macros and functions is the fact that macros do not have "runtime addresses" as functions do. This is due to the fact that preprocessor replaces the macro with the real contents before compilation. For these reasons macros cause slight overhead during compilation, but typically may execute faster than function calls.

For this very same reason you cannot easily replace an exported function in your own library with a macro having the same name and syntax. If you do so, you need to re-build all the applications that have been using that function.

All the preprocessor definitions are macros. Therefore, we could say that we heavily rely on them. However, there are quite remarkable differences in acceptable and unacceptable usage of macros.

© Neoxen Systems

Constants should always be defined to have descriptive names as described elsewhere in this guide. This is an acceptable approach and highly recommended.

For the sake of readability and maintainability, excessive use of custom macros as replacements for functions is usually not quite acceptable and is strongly discouraged.

You should not implement a macro which hides function exit points. This easily leads to maintenance problems when the code base grows in size.

There are some areas where macros are useful, but their usage should be well designed before implementation. For instance, if there is a message-oriented interface to certain functionality, it might be convenient to create a function style macro set for increasing the usability.

8.8 Memory Management

Current programming trends may not emphasize proper memory usage as much as they perhaps should. It is a misleading approach to consider memory usage optimization as a 'shadow from the past'.

8.8.1 Desktop Applications

For a standard desktop application, it might not be such a crucial issue, as long as the memory usage related bugs are eliminated. This category of applications is typically used only for a certain period of time before they are closed. Good programming practices and common sense are typically adequate for ensuring reasonable memory usage and performance.

8.8.2 System Software

For system components and serving modules of business applications this approach is typically inadequate. This kind of software should be able to run 24 hours a day and seven days a week without affecting the system's stability and responsiveness. If these applications are transaction intensive or serving a larger amount of concurrent users, the memory management requirement may become one of the most important aspects. Careless design can easily lead to memory fragmentation and decrease system robustness drastically.

© Neoxen Systems

8.8.3 General Recommendations

There is a performance difference with the standard memory management functions (malloc, calloc, realloc) and typically memory reallocation is recommended when appropriate. Windows API has also a set of memory management functions, typically suitable for client side programming.

As a general guideline, all allocated memory should be freed in the same place where it was allocated. Otherwise a proper allocate-free function pair should be implemented.

Neoxen QX Framework provides general memory management functions, which are recommended to be used whenever applicable. They are designed to be cross-platform compatible and optimized for efficiency and safety.

8.8.4 Storage Allocation

Static storage allocation is the most commonly used, simple and easy way for introducing buffers, such as strings. There are drawbacks however. If this approach is widely used, it may cause increase in maintenance expenses during the product life cycle. Also, if the storage sizes used are too small, they may limit the software’s usability. If the storage sizes are too large, memory usage can be highly inefficient. This scenario may cause memory fragmentation and scalability problems.

Dynamic storage allocation should be preferred in order to avoid the problems mentioned.

If static allocation is used, it is not recommended to use values, such as 256, directly. Should there later be a need to change a commonly used allocation size, it might be tedious to find all the correct occurrences to change. If some are missed, a possible bug is introduced.

Luckily, there is a very simple solution to this problem. All such constants should have a human readable definition in an appropriate header file.

© Neoxen Systems

```
#define QXGGC_MYMAXLEN   256

QXCHAR chString[QXGGC_MYMAXLEN] = '';


HGLOBAL hMem        = NULL;
LPTSTR  lptstrBuff  = NULL;

hMem = QxMySpecialAlloc(GPTR, QXGGC_MYMAXLEN);
if (!hMem)
{
   return QXGGC_ERR_ALLOC;
}
```

```
QxMyGetItemText( hwnd,
                 IDC_EDIT,
                (LPTSTR)chText,
                 sizeof(chText));
```

Note: In sample above sizeof() is used instead of hard coding. It is recommended to avoid direct constant usage wherever possible.

© Neoxen Systems

9 Compiler and Linker Warnings

When Visual Studio is used as the preferred development environment, compiler warning level should be kept at level 3. All the compilations with this level should be kept clean. This means that no warnings during compilation or linking should be allowed.

It is good programming practice and good housekeeping to clean out all the warnings. Preventing unnecessary warnings increases the product quality and saves time in maintenance as some of the warnings actually reveal hidden bugs.

The 64-bit portability issues should be taken into serious concern, when using Visual Studio 2005. The ‘Detect 64-bit Portability Issues’ setting (/W64) should always be set to ‘Yes’ in the project C++ properties. If Visual Studio 2008, 2010, 2012, 2013 or 2015 is used, this definition should not be used. Since Visual Studio 2008 it is deprecated and causes a compilation warning.

9.1 Using Explicit Typecasting

To avoid typical warnings of incompatible parameter types and in order to ensure proper byte alignment, explicit typecasting is highly recommended. Microsoft’s 64-bit compiler treats most of the missing typecasts as errors. Typecasting is the preferred method to direct the internal code optimizer for proper byte alignment for different data types.

9.2 Using VOID Parameter with C Compiler

If the project is compiled with the C-compiler instead of C++, the following function prototype is strictly forbidden. This is also the case if the function is exported from C++ under extern “C” modifier.

```
DWORD QXAPI Qxv_MyFunction();
```

Instead:

```
DWORD QXAPI Qxv_MyFunction(VOID);
```

© Neoxen Systems

This is due to limitations in parameter validation. If the function prototype is of the previous type, it can be called with parameters without compiler warnings. This may lead to mysterious error situations.

9.3 Using #pragma Directives

The typical mechanism to prevent some warnings in purpose is using preprocessor directives, such as *#pragma*. This is a useful mechanism, if used carefully, and in well-documented special cases only. If it is utilized loosely to hide peculiar coding practices, it is very likely to cause the increase of maintenance costs in the future.

© Neoxen Systems

10 Using Resource Files

10.1 Resource Symbols

Windows API still internally handles resource item identifiers as strings, i.e. names. Even though it is possible, it is not recommended for programmers to use resource names. The resource items should be given an environment independent resource ID as an integer value.

10.2 Version Information

According to professional Windows programming guidelines, each binary module should have a Version Info resource block. Appropriately designed contents of the block with product names, descriptions, version numbers, trademarks and copyright notices clearly indicate the professionalism of the vendor.

10.2.1 Traditional Approach

Visual Studio has an integrated and easy to use resource editor. Creating and maintaining resources, including version information is smooth and simple. It is just a perfect tool for small projects. However, there is a drawback when the project grows and the amount of modules increases.

In larger software projects, the manual maintenance of version information can become cumbersome, time consuming and error prone. Professional software build process needs more advanced mechanisms.

10.2.2 Neoxen QX Framework Approach

In order to enable automated build processes and minimize effort and human errors, Neoxen QX Framework offers an extension to the traditional management of version information blocks.

The project templates delivered with Neoxen QX Framework contain additional resource files for this purpose. The mechanism provided is fully compliant with Microsoft tools and it offers an intelligent option to centralize all version information of all modules in a single header file. Please refer to *Neoxen QX Framework technical documentation* for further details.

© Neoxen Systems

11 Commenting Source Code

There are many good practices for commenting source code. Whatever the guidelines are, they are appropriate, if they are reasonably adequate, consistent and systematically used.

The following chapters describe the commenting style and guidelines used in Neoxen QX Framework and the included templates and samples.

11.1 What to Comment?

11.1.1 File and Block Comments

All source code should be commented in the file level and in the functional level. Each file, except the files maintained by the development environment, should start with a descriptive comment block. Each function, class, method, etc., should have its own descriptive comment block as well.

In Neoxen QX Framework all the following files have their own variation of file level comment blocks:

- Source code files
- Header files
- Module definition files
- Make files
- Version Info resource files

Also the following have their own variation of comment block:

- Functions & Methods
- Class declarations

The variation between comment block types is minimal as far as formatting is concerned. Otherwise the content reflects the purpose.

Note: Please refer to the documentation provided with the Neoxen QX Framework templates for up-to-date information on the comment block contents.

© Neoxen Systems

11.1.2 Implementation Comments

This category of commenting has two variations. In the header files there should also be short comments separating logical entities, but the main usage should be functionality related.

Implementation comments are used within functions and methods. Also the comments describing variables, structure or class members and other similar descriptions belong to this category.

Comments should give some real value to the reader. It is better to leave comments out, than write useless or misleading descriptions. The best approach is to write short comments describing the logic, special conditions and exceptions to the established coding guidelines. Discipline and common sense are the key elements.

Note: Please refer to the documentation provided with the Neoxen QX Framework templates for up-to-date information and samples of implementation comments.

© Neoxen Systems

12 Naming Convention

Naming Conventions used in traditional UNIX programming are typically different compared to Windows. Linux/UNIX developers tend to use lower case without variable prefixing, C standard library style functions, lower case data types, etc. Native Windows code tends to use different naming convention. Variation of Hungarian notation is preferred, data types are capitalized, function and variable names are long and descriptive in mixed case, etc.

Too often it can be noticed that Linux/UNIX programmers disapprove the Windows coding style and Windows developers deprecate traditional Linux/UNIX coding style. Nevertheless, there are good aspects in both. The fact is that any coding style and naming convention can be good, as long as it respects professional coding practices, is consistent and is systematically used.

Due to its origin Neoxen QX Framework follows more closely the Windows coding style and naming conventions with some systematic modifications. The following chapters describe it in more detail.

12.1 Hungarian Notation

All the variable declarations in the header and source files are recommended to follow Neoxen QX Framework variation of the Hungarian notation. This means, that the variable names should start with a lower case prefix, which is derived from the data type of the variable.

There are good reasons to favor Hungarian notation. If it is used systematically, it can assist in producing robust code. Unintentional unsigned/signed mismatch problems as well as questionable assignments, comparisons and suspicious type casts can easily be seen from the code.

12.1.1 Simple Data Types

Hungarian notation is easy to use with simple data types. The prefix should be followed with the variable body starting with a capital letter. For example, variable of type unsigned long could be named:

```
ULONG ulMaxValue = QXGGC_MAXLIMIT;
```

© Neoxen Systems

The variable body should be descriptive of the context. Using the traditional single letter index variables (i, j, k, etc.) is not encouraged.

Please refer to *Appendix II* for a list of recommended prefixes.

12.1.2 Complex Data Types

Neoxen QX Framework naming convention is not so strict with complex data types. When naming variables of complex data types, the data type name or an abbreviation of it in lower case can be used either as a prefix or as the entire variable name, like:

```
MSG msg
PAINTSTRUCT ps
RECT rectUpd
```

Note: Development in Windows with C++ and MFC, as well as with C#, should follow the Microsoft proposed naming convention.

12.1.3 Using Data Type Definitions

Structures and unions should always be defined as complex data types. This allows much easier code maintenance in the future.

Following format is recommended:

```
typedef struct _QXMYSTRUCT
{
   HWND hwndMain;
   LPTSTR lptstrTitle;
}QXMYSTRUCT, *LPQXMYSTRUCT;
```

Note: Please notice that data types should always be defined with CAPITAL LETTERS.

12.2 Naming String Pointers

String pointers have their default prefixes according to their actual data type. It is strongly recommended to use precise prefixes according to the exact data type. This is especially useful, if the data type relates to UNICODE or has a constant modifier.

Declaration	Explanation
LPSTR lpstrText	Pointer to a null-terminated ANSI string
LPCSTR lpcstrText	Pointer to a constant null-terminated ANSI string
LPTSTR lptstrText	Pointer to a null-terminated ANSI or Unicode string
LPCTSTR lpctstrText	Pointer to a constant null-terminated ANSI or Unicode string
LPWSTR lpwstr	Pointer to a null-terminated Unicode string
LPCWSTR lpcwstrText	Pointer to a constant null-terminated Unicode string

Note: LPTSTR and LPCTSTR are the default data types to be used. Always prefer the string data types defined or recommended in the Neoxen QX Framework.

12.3 Naming Functions and Methods

Function and method names should be descriptive, in mixed (Camel) case and long enough to be as self-explanatory as possible.

Naming convention should be standardized and consistent through the whole code base. Names of the functions or methods should be unified within each module.

12.3.1 Using Module Prefixes

Each module should have its own unique two to four letter prefix and all the functions/methods within a module should use this same prefix consistently.

Sample:

```
AxgMyFunction()
AxgYourFunction()
AxgOtherFunction()
```

12.3.2 Using Logical Groups

The method and function names should be created so that they are logically grouped forming a consistent set of names. In practice this means that the target part of the name (object) is given first and then the action part (verb)

Sample:

```
AxgLogicalTreeCreate
AxgLogicalTreeGetProperties
AxgLogicalTreeSetProperties
AxgLogicalTreeDelete
```

12.4 Naming Constants

Constants should be defined with descriptive names, which are always in UPPER CASE. Underscores are allowed and recommended if readability so requires.

Constant definitions should be grouped in the header files with appropriate comments:

```
/*****************************************************************
* GENERAL STORAGE LIMITS
*****************************************************************/
#define AXG_MAXLEN                              256
#define AXG_MAX_NET_BUFF_SIZE                   1024
```

© Neoxen Systems

13 Exporting Functions

13.1 General Guidelines

When a module is designed, there should be a consistent approach on what to export and how. What should be accessible from outside and what should be kept internal? There should not be any reasons to allow access to internal data directly. It should be encapsulated inside the module and all manipulation should be done via function calls. This approach is quite object-oriented and recommended regardless of the programming language used.

13.2 How to Export?

There are some variations on how to export functions and variables from a module. You can export from both executables and libraries. Some of these mechanisms are Microsoft specific and are not recommended. The generally accepted standard mechanisms should be favored.

All the variations described in the following chapters can be used within the same module. However, it is recommended to standardize into one mechanism and use it consistently in all the development projects.

13.2.1 Using Microsoft Specific Storage-class Modifiers

The **dllimport** and **dllexport** storage-class modifiers are Microsoft-specific extensions to the C language and not recommended. However, these modifiers explicitly define the library interface to its client. Declaring functions as dllexport eliminates the need for a module-definition (.DEF) file. These modifiers can also be used with data and objects.

The dllimport and dllexport storage-class modifiers must be used with the extended attribute syntax keyword, __declspec:

© Neoxen Systems

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllExport int j;
DllExport int n;
```

13.2.2 Using Module Definition Files

A module definition (.DEF) file is a text file that contains statements for defining an executable or a library. For standard executables there is typically no need for creating such a file, as the linker provides equivalent command-line options for most module definition statements.

However, exporting functions via a module definition file is the mechanism recommended with Neoxen QX Framework programming.

In the exports sections of a module definition file, the functions can be exported by name and by ordinal number. Exporting by name is the most often used method, but there are benefits in adding the ordinals as well. The ordinals are also unique identifiers and another mechanism for locating the functions.

13.2.3 Using /EXPORT Specification

The exported functions can also be defined with a command line switch given to the linker. Even though supported, this is perhaps the most rarely used mechanism. Typically, there are no particular reasons to favor it.

Caution: Whether you export by name or by ordinal, you need to understand one fundamental aspect. Once published, you should never change the name or ordinal. If you do, the existing applications may cease functioning, as they cannot locate the function they are looking for.

© Neoxen Systems

13.2.4 Calling Conventions for Exported Functions

It is a good programming practice to use an explicit calling convention, it is especially important with exported functions. This approach guarantees proper argument handling and stack adjustment.

You are strongly advised to declare your exported functions explicitly as C-style functions using **extern "C"** modifier. This approach guarantees your code is accessible from almost any language capable of utilizing library calls.

Neoxen QX Framework consistently uses the QXAPI modifier, which currently defaults to _stdcall. Modifiers such as _stdcall should not be used directly. It is highly recommended to use definitions to abstract the actual calling convention.

With QXAPI modifier the arguments are guaranteed to be pushed from right to left, an underscore is prefixed to the name and the called functions do some argument checking during link time. The functions also do their own stack adjustment, when they return to caller.

There are some exceptions when direct declaration with **_cdecl** can be used. For instance, in case of vararg function **_cdecl** is always used, because the caller must clean up the parameter stack.

Note:	Library functions should not be attributed as _fastcall. Microsoft does not guarantee that the _fastcall modifier remains the same between compiler releases.

© Neoxen Systems

13.3 Avoiding Multiple Header Inclusion

It is good programming practice and good housekeeping to prevent the same header files being included multiple times during compilation. There is a simple mechanism to accomplish this.

The contents of a header file should be placed inside a following condition:

```
#ifndef __THISHEADER_H__
#define __THISHEADER_H__

//contents of this header

#endif    //__THISHEADER_H__
```

Typically, the condition marker (__THISHEADER_H__ in the sample above) is named according to the name of the header file. For instance, if the file name is PRINTDOC.H, the corresponding marker could be __PRINTDOC_H__.

© Neoxen Systems

14 Solving Problems

14.1 Switching from Debug to Release Build

Typically, most programmers develop and test their project in debug mode. After they consider the software to be adequately ready they create a release build and retest. It is far too common, that this is the point where the actual detective work starts. The debug version seemed to work nicely, but the release version crashes with access violations.

The father of REXX programming language, Mike Cowlishaw from IBM has said that the first question he used to ask from a young developer was the experience and capability in debugging. If the answer from the young interviewee was “good” or “very good”, Mr. Cowlishaw tended to consider the person as a bad programmer.

This may sound like exaggeration, but there is a valid point. Unfortunately debug mode seems to encourage some developers to loose programming practices, like extensive and careless ASSERT usage. The debug version is very forgiving and kind, but the release build is not.

The list below shows the primary differences between a debug and a release build. There are other differences, but the following are the primary differences that would cause an application to fail in a release build when it works in a debug build.

- Heap Layout
- Compilation
- Code Optimization
- Uninitialized Pointers

© Neoxen Systems

14.1.1 Heap Layout

Heap layout is said to cause about ninety percent of the apparent problems when an application works in a debug, but not in a release build.

When a project is built for debug, it is using a special debug memory allocation scheme. This means that all memory allocations have guard bytes placed around them. These guard bytes are placed in order to discover memory overwrites. As the heap layout is different between release and debug versions, a memory overwrite might not create any problems in a debug build, but may have fatal consequences in a release build.

When a call to a heap manipulation function causes an access violation, it is possible that the program has corrupted the heap. A common symptom of this situation is "Access Violation in _searchseg".

Note: The _heapchk function is available in both debug and release builds for verifying the integrity of the run-time library heap. This function can be used to isolate a heap overwrite.

14.1.2 Compilation

Microsoft Foundation Classes (MFC) is widely used in C++ programming and it is often used together with Neoxen QX Framework as well. However, there are some issues to be aware of. Many of the macros and much of the actual implementation in MFC changes when you create a release build. In particular, the ASSERT macro evaluates to nothing in a release build; so none of the code found in ASSERT statements will ever be executed.

Careless usage of ASSERT macros may easily cause a release version of an MFC application to crash, return incorrect results, or exhibit some other abnormal behavior.

This problem can be caused when important code is placed in an ASSERT statement to verify correct functionality. Because ASSERT statements are commented out in a release build of an MFC application, the code does not run in a release build.

© Neoxen Systems

One proven technique to avoid this problem is to assign the function's return value to a temporary variable and then test the variable in an ASSERT statement.

When peculiar behavior is encountered, it may originate from code optimization. For instance, some functions are inlined for increased speed in the release build. Optimizations are generally turned on in a release build and a different memory allocation schema is also being used.

Note: If you are using ASSERT to confirm that a function call succeeded, consider using VERIFY instead. The VERIFY macro evaluates its own arguments in both debug and release builds.

14.1.3 Code Optimization

Depending on the nature of certain segments of code, the optimizing compiler might generate unexpected code. Occasionally this may cause instability or peculiar behavior especially in a release build.

During optimization, the compiler reorganizes and repositions instructions generated from the source code, resulting in more efficient execution of the application. Because the structure after optimization is rearranged, the debugger cannot always identify the correct piece of source code that corresponds to a set of instructions. Therefore, if problems occur, it is advisable to debug the code before optimizing it. Optimization can then be re-enabled after debugging.

If there is a reason to suspect that a particular portion of the code is not being optimized correctly and is causing problems in the compilation or execution of the application, the offending code can be bracketed with

```
#pragma optimize("", off)

// Here is the code to investigate

#pragma optimize("", on)
```

© Neoxen Systems

14.1.4 Uninitialized Pointers

The lack of debugging information removes the padding from your application. In a release build, stray pointers have a greater chance of pointing to uninitialized memory instead of pointing to debug information.

© Neoxen Systems

Index

Adjust Processes 10
APIENTRY...............................61
Archiving................................22
ASSERT............................ 63, 64
Backup..................................22
Backwards Compatible..............10
Bandwidth 17
Better Applications 10
Bug Tracking 9
Build Engine 4, 25, 26
Build Machine 4, 25
Build Master 4, 25
Build Number..........................26
Build Process 4, 8, 25, 26
Business Applications...............13
Business Level Functionality ..9, 10
Business solutions 11
C/C++ 7, 24, 31, 50
C++ 10, 36, 46, 56, 64
Caching Algorithms 20
C-language............10, 40, 46, 59
Clean Environment 4, 25
Client/Server 20
CMS........................4, 24, 25, 26
Communications 12
Competitive Advantage 10
Components 11, 13, 15, 18, 21, 23
Concurrent Connections 11
Configuration 14
Configuration Management8, 24, 25, 29
Consistent Interface 9
Constructor 21
C-style Functions 2
C-Style Functions 10
Customer Care Services 9
Customer Commitments...........26
Customer Satisfaction 10
Customizable 9
CVS 4, 29
Data Initialization.....................21
Data Refreshing 21
Data Transportation 15
Database 12
Databases 19
Designing Robust Software 11
Desktop Applications 11
Development Costs 9
Distributed Systems 11
dllexport 59
dllimport 59
DOS 40, 55
Extern C 61
Globalization 13
Gnu Make 26
Hard Tabs 32
Hidden Costs 8, 9
HTTP 4, 17
Hungarian Notation 55
Increase Efficiency 10
Internationalization 10
Internet 15
Introduction 6
Java 10
JRE.................................. 4, 12
Just-in-time Reload 21
LAN 4, 15
Licensing 9
Linux 12, 24, 25, 26, 29
Load Balancing 18
Localization 10, 40
Macro Languages 10
Make-file Hierarchy 25
Messaging Solutions 17
MFC.................. 4, 30, 31, 56, 64
Microsoft......................24, 56, 59
Modular Architecture 9
Module Definition File 60
Multi-layered Architecture 9
Multiprocessing 37
Multithreading 37
multi-tier programming 2
Native Layer 12
Neoxen 5

© Neoxen Systems

Nmake 26
Object Model 19
OS/2......................................55
Output Directory 28
Persistent Connection 17
Platform SDK 7
Predefined Algorithms...............18
Presentation Manager 55
Product Management 9
Project Completion 10
Project Costs 9
Project Management 9
Project Workspace 30
Protocol Stack..........................12
Proven Foundation....................10
Quality Assurance 9
Rapid Development 11
Refine Processes 10
related documentation 5
Reliability 11
Request Queue 13
Robustness.............................11
ROI 4, 15
Royalty-free 9
Scalability 18
Scripting Languages 10
Secure Network........................15
SEH 4, 46
Server....................................11
Software Build 24
Source Code Tree.......................8
SQL 4
SQL Statement 19
Stream Socket.........................13
Stream Sockets 19
Subcontractors 28
Support Costs 10
System Resources 13
Technical Requirements............11
Technical Support 9
Terms and concepts 4
abbreviations 4
terminology 4
Testing 12
Testing Cycle...........................10
Third Party 12, 18, 22, 23
Threads and Processes..............13
Try-Catch 45
Typecasting 50
Unix 40, 55
VERIFY 65
Version Control System 25
Visual Basic 56
Visual Studio24, 26, 28, 30, 32, 50, 52
WAN 4, 15
Web Browser 12
Web Server 12
Windows . 7, 12, 29, 48, 52, 55, 56
Windows API 52

APPENDIX I: Data Types

ANSI C-style simple data types should not be used, if not necessary. For maximum compliance between existing and forthcoming 32/64-bit versions of Microsoft Windows, it is highly recommended to use the data types introduced in the Neoxen QX Framework.

Please refer to the Technical Documentation to see the up-to-date list of recommended data types.

Note:	If there is a data type defined in Neoxen QX Framework header files, you should always use it.

© Neoxen Systems

APPENDIX II: Data Type Prefixes

All the variable declarations in the header and source files should follow the Neoxen QX Framework variation of the Hungarian notation. This means, that all the variable names should start with a lower case prefix, which is derived from the data type of the variable.

Please refer to the Technical Documentation to see the up-to-date list of recommended data types and respective prefixes.

Note: If there is a recommended prefix for a data type defined, you are strongly encouraged to use it.

© Neoxen Systems

DOCUMENTATION LICENSE

This documentation, as well as the software described in it, is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Neoxen Systems.

Neoxen Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Neoxen Systems.

Information in this document is provided in connection with the vendor products. No license, express or implied, by estoppels or otherwise, to any intellectual property rights is granted by this document.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Neoxen Systems reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

TRADEMARKS

Neoxen, the Neoxen logo, Trelox, Neoxen QX, Neoxen QX Framework, Neoxen Modus, Neoxen Visual Modus, Neoxen iModus, Neoxen EveryPlace and Neoxen NaviList are trademarks or registered trademarks of Neoxen Systems in USA and/or other countries.

Microsoft, Microsoft Office, Microsoft Windows and Microsoft Visual Studio are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Adobe, Acrobat and Acrobat Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Linux is a registered trademark of Linus Torvalds.

All other trademarks, registered trademarks and/or product names are property of their respective owners.

COPYRIGHT

Template from Neoxen Modus Methodology, copyright © 2016 Neoxen Systems.
© 2016 Neoxen Systems. All Rights Reserved.

RESTRICTED RIGHTS LEGENDS

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software -- Restricted Rights in 48 CFR 52.227-19 as applicable.

Unpublished - rights reserved under the Copyright Laws of the United States and International treaties.

http://www.neoxen.com

sales@neoxen.com