unicode.cm

```
/*
    ========================================================================

    Copyright (c) 2012-2016 Seppo Laakko
    http://sourceforge.net/projects/cmajor/

    Distributed under the GNU General Public License, version 3 (GPLv3).
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html
        )

 ========================================================================
    */

using System;
using System.Collections;
using System.IO;

namespace System.Unicode
{
    public enum Category : uint
    {
        none = 0u,
        letterUpper = 1u << 0u,                   // Lu
        letterLower = 1u << 1u,                   // Ll
        letterCased = 1u << 2u,                   // LC
        letterModifier = 1u << 3u,                // Lm
        letterOther = 1u << 4u,                   // Lo
        letterTitle = 1u << 5u,                   // Lt
        letter = letterUpper | letterLower | letterCased | letterModifier
            | letterOther | letterTitle,
        markSpacing = 1u << 6u,                   // Mc
        markEnclosing = 1u << 7u,                 // Me
        markNonspacing = 1u << 8u,                // Mn
        mark = markSpacing | markEnclosing | markNonspacing,
        numberDecimal = 1u << 9u,                 // Nd
        numberLetter = 1u << 10u,                 // Nl
        numberOther = 1u << 11u,                  // No
        number = numberDecimal | numberLetter | numberOther,
        punctuationConnector = 1u << 12u,         // Pc
        punctuationDash = 1u << 13u,              // Pd
        punctuationClose = 1u << 14u,             // Pe
        punctuationFinalQuote = 1u << 15u,        // Pf
        punctuationInitialQuote = 1u << 16u,      // Pi
        punctuationOther = 1u << 17u,             // Po
        punctuationOpen = 1u << 18u,              // Ps
        punctuation = punctuationConnector | punctuationDash |
            punctuationClose | punctuationFinalQuote |
            punctuationInitialQuote | punctuationOther | punctuationOpen,
        symbolCurrency = 1u << 19u,               // Sc
        symbolModifier = 1u << 20u,               // Sk
        symbolMath = 1u << 21u,                   // Sm
    symbolOther = 1u << 22u,                  // So
    symbol = symbolCurrency | symbolModifier | symbolMath |
        symbolOther,
    separatorLine = 1u << 23u,                // Zl
    separatorParagraph = 1u << 24u,           // Zp
    separatorSpace = 1u << 25u,               // Zs
    separator = separatorLine | separatorParagraph | separatorSpace
}

public nothrow ulong GetHashCode(Category category)
{
    return cast<ulong>(category);
}

public class CategoryMap
{
    static CategoryMap() : instance(new CategoryMap())
    {
    }
    public nothrow static CategoryMap& Instance()
    {
        #assert(!instance.IsNull());
        return *instance;
    }
    public CategoryMap()
    {
        strCategoryMap["Lu"] = Category.letterUpper;
        categoryStrMap[Category.letterUpper] = "Lu";
        strCategoryMap["Ll"] = Category.letterLower;
        categoryStrMap[Category.letterLower] = "Ll";
        strCategoryMap["LC"] = Category.letterCased;
        categoryStrMap[Category.letterCased] = "LC";
        strCategoryMap["Lm"] = Category.letterModifier;
        categoryStrMap[Category.letterModifier] = "Lm";
        strCategoryMap["Lo"] = Category.letterOther;
        categoryStrMap[Category.letterOther] = "Lo";
        strCategoryMap["Lt"] = Category.letterTitle;
        categoryStrMap[Category.letterTitle] = "Lt";
        strCategoryMap["Mc"] = Category.markSpacing;
        categoryStrMap[Category.markSpacing] = "Mc";
        strCategoryMap["Me"] = Category.markEnclosing;
        categoryStrMap[Category.markEnclosing] = "Me";
        strCategoryMap["Mn"] = Category.markNonspacing;
        categoryStrMap[Category.markNonspacing] = "Mn";
        strCategoryMap["Nd"] = Category.numberDecimal;
        categoryStrMap[Category.numberDecimal] = "Nd";
        strCategoryMap["Nl"] = Category.numberLetter;
        categoryStrMap[Category.numberLetter] = "Nl";
        strCategoryMap["No"] = Category.numberOther;
        categoryStrMap[Category.numberOther] = "No";
        strCategoryMap["Pc"] = Category.punctuationConnector;
        categoryStrMap[Category.punctuationConnector] = "Pc";
        strCategoryMap["Pd"] = Category.punctuationDash;
```

```
            categoryStrMap[Category.punctuationDash] = "Pd";
            strCategoryMap["Pe"] = Category.punctuationClose;
            categoryStrMap[Category.punctuationClose] = "Pe";
            strCategoryMap["Pf"] = Category.punctuationFinalQuote;
            categoryStrMap[Category.punctuationFinalQuote] = "Pf";
            strCategoryMap["Pi"] = Category.punctuationInitialQuote;
            categoryStrMap[Category.punctuationInitialQuote] = "Pi";
            strCategoryMap["Po"] = Category.punctuationOther;
            categoryStrMap[Category.punctuationOther] = "Po";
            strCategoryMap["Ps"] = Category.punctuationOpen;
            categoryStrMap[Category.punctuationOpen] = "Ps";
            strCategoryMap["Sc"] = Category.symbolCurrency;
            categoryStrMap[Category.symbolCurrency] = "Sc";
            strCategoryMap["Sk"] = Category.symbolModifier;
            categoryStrMap[Category.symbolModifier] = "Sk";
            strCategoryMap["Sm"] = Category.symbolMath;
            categoryStrMap[Category.symbolMath] = "Sm";
            strCategoryMap["So"] = Category.symbolOther;
            categoryStrMap[Category.symbolOther] = "So";
            strCategoryMap["Zl"] = Category.separatorLine;
            categoryStrMap[Category.separatorLine] = "Zl";
            strCategoryMap["Zp"] = Category.separatorParagraph;
            categoryStrMap[Category.separatorParagraph] ="Zp";
            strCategoryMap["Zs"] = Category.separatorSpace;
            categoryStrMap[Category.separatorSpace] = "Zs";
        }
        public nothrow Category GetCategory(const string& categoryName)
            const
        {
            HashMap<string, Category>.ConstIterator i = strCategoryMap.
                CFind(categoryName);
            if (i != strCategoryMap.CEnd())
            {
                return i->second;
            }
            return Category.none;
        }
        public nothrow string GetCategoryName(Category category) const
        {
            HashMap<Category, string>.ConstIterator i = categoryStrMap.
                CFind(category);
            if (i != categoryStrMap.CEnd())
            {
                return i->second;
            }
            return "";
        }
        private HashMap<string, Category> strCategoryMap;
        private HashMap<Category, string> categoryStrMap;
        private static UniquePtr<CategoryMap> instance;
    }

    public class CharacterInfo
```

```
{
    public CharacterInfo() : code(cast<uchar>(0u)), name(), category(
        Category.none), toLower(cast<uchar>(0u)), toUpper(cast<uchar
        >(0u))
    {
    }
    public CharacterInfo(uchar code_, const string& name_, Category
        category_, uchar toLower_, uchar toUpper_) : code(code_), name
        (name_), category(category_), toLower(toLower_), toUpper(
        toUpper_)
    {
    }
    public nothrow inline uchar Code() const
    {
        return code;
    }
    public nothrow inline const string& Name() const
    {
        return name;
    }
    public nothrow inline Category GetCategory() const
    {
        return category;
    }
    public nothrow inline uchar ToLower() const
    {
        return toLower;
    }
    public nothrow inline uchar ToUpper() const
    {
        return toUpper;
    }
    public nothrow inline bool IsLetter() const
    {
        return (category & Category.letter) != Category.none;
    }
    public nothrow inline bool IsMark() const
    {
        return (category & Category.mark) != Category.none;
    }
    public nothrow inline bool IsNumber() const
    {
        return (category & Category.number) != Category.none;
    }
    public nothrow inline bool IsPunctuation() const
    {
        return (category & Category.punctuation) != Category.none;
    }
    public nothrow inline bool IsSymbol() const
    {
        return (category & Category.symbol) != Category.none;
    }
    public nothrow inline bool IsSeparator() const
    {
        return (category & Category.separator) != Category.none;
    }
    public void Read(BinaryFileStream& unicodeBin)
    {
        code = cast<uchar>(unicodeBin.ReadUInt());
        name = unicodeBin.ReadString();
        category = cast<Category>(unicodeBin.ReadUInt());
        toLower = cast<uchar>(unicodeBin.ReadUInt());
        toUpper = cast<uchar>(unicodeBin.ReadUInt());
    }
    public void Write(BinaryFileStream& unicodeBin)
    {
        unicodeBin.Write(cast<uint>(code));
        unicodeBin.Write(name);
        unicodeBin.Write(cast<uint>(category));
        unicodeBin.Write(cast<uint>(toLower));
        unicodeBin.Write(cast<uint>(toUpper));
    }
    private uchar code;
    private string name;
    private Category category;
    private uchar toLower;
    private uchar toUpper;
}

public class CharacterInfoMap
{
    public static void Load()
    {
        if (instance.IsNull())
        {
            instance.Reset(new CharacterInfoMap(true));
        }
    }
    public static void Construct()
    {
        if (instance.IsNull())
        {
            instance.Reset(new CharacterInfoMap(false));
        }
    }
    private CharacterInfoMap(bool read)
    {
        if (read)
        {
            string unicodeBinFilePath = Path.Combine(
                PathToSystemDirectory(), "unicode.bin");
            BinaryFileStream unicodeBin(unicodeBinFilePath, OpenMode.
                readOnly);
            Read(unicodeBin);
        }
    }
    public static CharacterInfoMap& Instance()
    {
        #assert(!instance.IsNull());
        return *instance;
    }
    public void Read(BinaryFileStream& unicodeBin)
    {
        uint n = unicodeBin.ReadUInt();
        for (uint i = 0u; i < n; ++i)
        {
            CharacterInfo* info = new CharacterInfo();
            info->Read(unicodeBin);
            infos.Add(UniquePtr<CharacterInfo>(info));
            map[info->Code()] = info;
        }
    }
    public void Write(BinaryFileStream& unicodeBin)
    {
        uint n = cast<uint>(infos.Count());
        unicodeBin.Write(n);
        for (uint i = 0u; i < n; ++i)
        {
            infos[cast<int>(i)]->Write(unicodeBin);
        }
    }
    public void Add(CharacterInfo* characterInfo)
    {
        infos.Add(UniquePtr<CharacterInfo>(characterInfo));
    }
    public CharacterInfo* GetCharacterInfo(uchar c) const
    {
        HashMap<uchar, CharacterInfo*>.ConstIterator i = map.CFind(c)
            ;
        if (i != map.CEnd())
        {
            return i->second;
        }
        return null;
    }
    private static UniquePtr<CharacterInfoMap> instance;
    private HashMap<uchar, CharacterInfo*> map;
    private List<UniquePtr<CharacterInfo>> infos;
}

public Category GetCategory(uchar c)
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
        GetCharacterInfo(c);
    if (info != null)
    {
        return info->GetCategory();
    }
```

```
        return Category.none;
    }

    public nothrow string GetCategoryName(Category category)
    {
        return CategoryMap.Instance().GetCategoryName(category);
    }

    public string GetCharacterName(uchar c)
    {
        CharacterInfoMap.Load();
        CharacterInfo* info = CharacterInfoMap.Instance().
            GetCharacterInfo(c);
        if (info != null)
        {
            return info->Name();
        }
        return "";
    }

    public uchar ToLower(uchar c)
    {
        CharacterInfoMap.Load();
        CharacterInfo* info = CharacterInfoMap.Instance().
            GetCharacterInfo(c);
        if (info != null)
        {
            uchar toLower = info->ToLower();
            if (toLower != cast<uchar>(0u))
            {
                return toLower;
            }
        }
        return c;
    }

    public uchar ToUpper(uchar c)
    {
        CharacterInfoMap.Load();
        CharacterInfo* info = CharacterInfoMap.Instance().
            GetCharacterInfo(c);
        if (info != null)
        {
            uchar toUpper = info->ToUpper();
            if (toUpper != cast<uchar>(0u))
            {
                return toUpper;
            }
        }
        return c;
    }

    public bool IsLetter(uchar c)
```

```
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
       GetCharacterInfo(c);
    if (info != null)
    {
        return info->IsLetter();
    }
    return false;
}

public bool IsLower(uchar c)
{
    return GetCategory(c) == Category.letterLower;
}

public bool IsUpper(uchar c)
{
    return GetCategory(c) == Category.letterUpper;
}

public bool IsMark(uchar c)
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
       GetCharacterInfo(c);
    if (info != null)
    {
        return info->IsMark();
    }
    return false;
}

public bool IsNumber(uchar c)
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
       GetCharacterInfo(c);
    if (info != null)
    {
        return info->IsNumber();
    }
    return false;
}

public bool IsPunctuation(uchar c)
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
       GetCharacterInfo(c);
    if (info != null)
    {
        return info->IsPunctuation();
    }
    return false;
}

public bool IsSymbol(uchar c)
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
       GetCharacterInfo(c);
    if (info != null)
    {
        return info->IsSymbol();
    }
    return false;
}

public bool IsSeparator(uchar c)
{
    CharacterInfoMap.Load();
    CharacterInfo* info = CharacterInfoMap.Instance().
       GetCharacterInfo(c);
    if (info != null)
    {
        return info->IsSeparator();
    }
    return false;
}

public ustring ToUtf32(const string& utf8Str)
{
    ustring result;
    const char* p = utf8Str.Chars();
    int bytesRemaining = utf8Str.Length();
    while (bytesRemaining > 0)
    {
        char c = *p;
        byte x = cast<byte>(c);
        if ((x & 0x80u) == 0u)
        {
            result.Append(cast<uchar>(cast<uint>(x)));
            --bytesRemaining;
            ++p;
        }
        else if ((x & 0xE0u) == 0xC0u)
        {
            if (bytesRemaining < 2)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            uchar u = cast<uchar>(cast<uint>(0u));
            byte b1 = cast<byte>(p[1]);
            if ((b1 & 0xC0u) != 0x80u)
            {
```

```
                ThrowConversionException("invalid UTF-8 sequence");
            }
            byte shift = 0u;
            for (byte i = 0u; i < 6u; ++i)
            {
                byte bit = b1 & 1u;
                b1 = b1 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            byte b0 = x;
            for (byte i = 0u; i < 5u; ++i)
            {
                byte bit = b0 & 1u;
                b0 = b0 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            result.Append(u);
            bytesRemaining = bytesRemaining - 2;
            p = p + 2;
        }
        else if ((x & 0xF0u) == 0xE0u)
        {
            if (bytesRemaining < 3)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            uchar u = cast<uchar>(cast<uint>(0u));
            byte b2 = cast<byte>(p[2]);
            if ((b2 & 0xC0u) != 0x80u)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            byte shift = 0u;
            for (byte i = 0u; i < 6u; ++i)
            {
                byte bit = b2 & 1u;
                b2 = b2 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            byte b1 = cast<byte>(p[1]);
            if ((b1 & 0xC0u) != 0x80u)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            for (byte i = 0u; i < 6u; ++i)
            {
                byte bit = b1 & 1u;
                b1 = b1 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            byte b0 = x;
            for (byte i = 0u; i < 4u; ++i)
            {
                byte bit = b0 & 1u;
                b0 = b0 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            result.Append(u);
            bytesRemaining = bytesRemaining - 3;
            p = p + 3;
        }
        else if ((x & 0xF8u) == 0xF0u)
        {
            if (bytesRemaining < 4)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            uchar u = cast<uchar>(cast<uint>(0u));
            byte b3 = cast<byte>(p[3]);
            if ((b3 & 0xC0u) != 0x80u)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            byte shift = 0u;
            for (byte i = 0u; i < 6u; ++i)
            {
                byte bit = b3 & 1u;
                b3 = b3 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            byte b2 = cast<byte>(p[2]);
            if ((b2 & 0xC0u) != 0x80u)
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
            for (byte i = 0u; i < 6u; ++i)
            {
                byte bit = b2 & 1u;
                b2 = b2 >> 1u;
                u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                    shift));
                ++shift;
            }
            byte b1 = cast<byte>(p[1]);
```

```
                if ((b1 & 0xC0u) != 0x80u)
                {
                    ThrowConversionException("invalid UTF-8 sequence");
                }
                for (byte i = 0u; i < 6u; ++i)
                {
                    byte bit = b1 & 1u;
                    b1 = b1 >> 1u;
                    u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                        shift));
                    ++shift;
                }
                byte b0 = x;
                for (byte i = 0u; i < 3u; ++i)
                {
                    byte bit = b0 & 1u;
                    b0 = b0 >> 1u;
                    u = cast<uchar>(cast<uint>(u) | (cast<uint>(bit) <<
                        shift));
                    ++shift;
                }
                result.Append(u);
                bytesRemaining = bytesRemaining - 4;
                p = p + 4;
            }
            else
            {
                ThrowConversionException("invalid UTF-8 sequence");
            }
        }
        return result;
    }

    public ustring ToUtf32(const wstring& utf16Str)
    {
        ustring result;
        const wchar* w = utf16Str.Chars();
        int remaining = utf16Str.Length();
        while (remaining > 0)
        {
            wchar w1 = *w++;
            --remaining;
            if (cast<ushort>(w1) < 0xD800u || cast<ushort>(w1) > 0xDFFFu)
            {
                result.Append(w1);
            }
            else
            {
                if (cast<ushort>(w1) < 0xD800u || cast<ushort>(w1) > 0
                    xDBFFu)
                {
                    ThrowConversionException("invalid UTF-16 sequence");
                }
                if (remaining > 0)
                {
                    wchar w2 = *w++;
                    --remaining;
                    if (cast<ushort>(w2) < 0xDC00u || cast<ushort>(w2) >
                        0xDFFFu)
                    {
                        ThrowConversionException("invalid UTF-16 sequence
                            ");
                    }
                    else
                    {
                        uchar uprime = cast<uchar>(((0x03FFu & cast<uint
                            >(w1)) << 10u) | (0x03FFu & cast<uint>(w2)));
                        uchar u = cast<uchar>(cast<uint>(uprime) + 0
                            x10000u);
                        result.Append(u);
                    }
                }
                else
                {
                    ThrowConversionException("invalid UTF-16 sequence");
                }
            }
        }
        return result;
    }

    public wstring ToUtf16(const ustring& utf32Str)
    {
        wstring result;
        for (uchar u : utf32Str)
        {
            if (cast<uint>(u) > 0x10FFFFu)
            {
                ThrowConversionException("invalid UTF-32 code point");
            }
            if (cast<uint>(u) < 0x10000u)
            {
                if (cast<uint>(u) >= 0xD800 && cast<uint>(u) <= 0xDFFF)
                {
                    ThrowConversionException("invalid UTF-32 code point (
                        reserved for UTF-16)");
                }
                wchar x = cast<wchar>(u);
                result.Append(x);
            }
            else
            {
                uchar uprime = cast<uchar>(cast<uint>(u) - 0x10000u);
                wchar w1 = cast<wchar>(0xD800u);
                wchar w2 = cast<wchar>(0xDC00u);
                for (ushort i = 0u; i < 10u; ++i)
                {
                    ushort bit = cast<ushort>(cast<uint>(uprime) & (cast<
                        uint>(0x1u) << i));
                    w2 = cast<wchar>(cast<ushort>(w2) | bit);
                }
                for (ushort i = 10u; i < 20u; ++i)
                {
                    ushort bit = cast<ushort>((cast<uint>(uprime) & (cast
                        <uint>(0x1u) << i)) >> 10u);
                    w1 = cast<wchar>(cast<ushort>(w1) | bit);
                }
                result.Append(w1);
                result.Append(w2);
            }
        }
        return result;
    }

    public wstring ToUtf16(const string& utf8Str)
    {
        return ToUtf16(ToUtf32(utf8Str));
    }

    public string ToUtf8(const ustring& utf32Str)
    {
        string result;
        for (uchar c : utf32Str)
        {
            uint x = cast<uint>(c);
            if (x < 0x80u)
            {
                result.Append(cast<char>(x & 0x7Fu));
            }
            else if (x < 0x800u)
            {
                byte b1 = 0x80u;
                for (byte i = 0u; i < 6u; ++i)
                {
                    b1 = b1 | (cast<byte>(x & 1u) << i);
                    x = x >> 1u;
                }
                byte b0 = 0xC0u;
                for (byte i = 0u; i < 5u; ++i)
                {
                    b0 = b0 | (cast<byte>(x & 1u) << i);
                    x = x >> 1u;
                }
                result.Append(cast<char>(b0));
                result.Append(cast<char>(b1));
            }
            else if (x < 0x10000u)
            {
                byte b2 = 0x80u;
        for (byte i = 0u; i < 6u; ++i)
        {
            b2 = b2 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        byte b1 = 0x80u;
        for (byte i = 0u; i < 6u; ++i)
        {
            b1 = b1 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        byte b0 = 0xE0u;
        for (byte i = 0u; i < 4u; ++i)
        {
            b0 = b0 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        result.Append(cast<char>(b0));
        result.Append(cast<char>(b1));
        result.Append(cast<char>(b2));
    }
    else if (x < 0x110000u)
    {
        byte b3 = 0x80u;
        for (byte i = 0u; i < 6u; ++i)
        {
            b3 = b3 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        byte b2 = 0x80u;
        for (byte i = 0u; i < 6u; ++i)
        {
            b2 = b2 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        byte b1 = 0x80u;
        for (byte i = 0u; i < 6u; ++i)
        {
            b1 = b1 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        byte b0 = 0xF0u;
        for (byte i = 0u; i < 3u; ++i)
        {
            b0 = b0 | (cast<byte>(x & 1u) << i);
            x = x >> 1u;
        }
        result.Append(cast<char>(b0));
        result.Append(cast<char>(b1));
        result.Append(cast<char>(b2));
        result.Append(cast<char>(b3));
    }
    else
```

```
                {
                    ThrowConversionException("invalid UTF−32 code point");
                }
            }
            return result;
        }

        public string ToUtf8(const wstring& utf16Str)
        {
            return ToUtf8(ToUtf32(utf16Str));
        }

        public nothrow inline char SeparatorChar()
        {
        #if (WINDOWS)
            return ';';
        #endif
            return ':';
        }

        public string PathToSystemDirectory()
        {
            char* cmLibraryPath = get_environment_variable("CM_LIBRARY_PATH")
                ;
            string cmLibPath;
            if (cmLibraryPath != null)
            {
                cmLibPath = cmLibraryPath;
            }
            if (cmLibPath.IsEmpty())
            {
                throw Exception("CM_LIBRARY_PATH environment variable not set
                    ");
            }
            List<string> paths = cmLibPath.Split(SeparatorChar());
            if (paths.Count() > 0)
            {
                return paths[0];
            }
            else
            {
                throw Exception("library paths empty");
            }
        }
    }
```