

# Symbian OS Descriptors Cookbook

Alexey Gusev

Published by the Symbian Developer Network

Version: 1.0 – September 2008

<b>1 ABSTRACT.....</b>	<b>2</b>
<b>2 FOREWORD.....</b>	<b>2</b>
<b>3 EASY RECIPES.....</b>	<b>2</b>
3.1 HOW TO CONCATENATE TWO STRINGS.....	2
3.2 HOW TO FORMAT A NUMBER INTO A STRING.....	4
3.3 HOW TO CONVERT A DESCRIPTOR TO A NUMBER.....	4
3.4 HOW TO EXTERNALIZE/INTERNALIZE A DESCRIPTOR.....	5
3.5 HOW TO WRITE BINARY DATA INTO A DESCRIPTOR OR STRING LITERAL.....	6
<b>4 MORE COMPLEX RECIPES.....</b>	<b>6</b>
4.1 HOW TO FORMAT STRINGS.....	6
4.2 HOW TO CONVERT A C STRING TO A DESCRIPTOR (AND VICE VERSA).....	8
4.3 HOW TO CONVERT AN 8-BIT STRING TO A 16-BIT STRING (AND VICE VERSA).....	8
4.4 HOW TO PASS DESCRIPTORS AS FUNCTION ARGUMENTS.....	9
4.5 HOW TO PASS AND RETURN MODIFIABLE DESCRIPTORS.....	10
<b>5 ADVANCED RECIPES.....</b>	<b>10</b>
5.1 HOW TO CONVERT FROM ONE DESCRIPTOR TYPE TO ANOTHER.....	10
5.2 HOW TO USE RBUF WHEN A MODIFIABLE DYNAMIC DESCRIPTOR IS NEEDED.....	11
5.3 HOW TO FIND AND MATCH DESCRIPTORS.....	12
5.4 HOW TO COMPARE DESCRIPTORS.....	13
5.5 HOW TO USE TLEX.....	15
<b>6 CONCLUSION.....</b>	<b>16</b>
<b>7 AUTHOR PROFILE.....</b>	<b>17</b>

## 1 Abstract

This article illustrates the use of one of the more challenging entities in Symbian OS: descriptors. These string classes are frequently criticized by developers new to Symbian OS. Even the more seasoned engineer might have to spend some time to resolve common problems when working with the descriptors classes.

This article provides a clear and definite guide for everyone who wants to get handy experience with these classes, learn how they work and get brief solutions for the most common problems that a developer faces in his or her everyday job. The article is laid out like a cookbook, in a style similar to the recently published book, **Quick Recipes on Symbian OS** from Symbian Press (more information about the book can be found at [developer.symbian.com/quick](http://developer.symbian.com/quick)).

The recipe-based structure of the article ensures that each section may be studied separately, giving the reader a convenient and easy way to look for the solutions whenever he or she needs it.

All of the code snippets in the recipes highlight particular sections of the descriptors API, from simple string manipulations to the conversions between various types of descriptors and to lexical analysis.

The reader is expected to have some basic knowledge of C++. You should also have some knowledge of the descriptor class hierarchy because this article does not cover it in detail. You can find a discussion of this in the Symbian Developer Library documentation (available at [developer.symbian.com/main/documentation/sdl](http://developer.symbian.com/main/documentation/sdl)) and in Symbian Press publications (see more details about these at [developer.symbian.com/books](http://developer.symbian.com/books)).

## 2 Foreword

If you're not a fan of Symbian OS descriptors yet, then this article hopes to encourage you to change your mind. Before you get your hands dirty on the recipes below, let us mention just a couple of things to keep in mind while you are reading the article.

The main point to remember is that although the descriptors in Symbian OS are similar to the string classes in other operating systems or frameworks, the memory management is entirely the responsibility of the developer. This has great implications on the way you write your code, when you could expect it to 'leave' or 'panic,' and so on.

For the sake of simplicity and clarity, the code samples in this article do not include error-handling code. The accompanying sample project does use standard Symbian C++ techniques to ensure leave-safety and so it gives you a better idea of how to work with descriptors in practice.

Most of the recipes use some common declarations; these are only listed in the first code sample to avoid unnecessary duplication

This said, we are now ready to dive into the adventure and explore the world of Symbian OS descriptors.

## 3 Easy Recipes

### 3.1 How to Concatenate Two Strings

**The problem:** You have two strings and want to concatenate them.

**The solution:** If you have programmed using any other operating system in any object-oriented language, you could expect the solution to be as simple as using operator+(). The good news is

that on Symbian OS this works for descriptors. Well, almost. You should use `TDesC::operator+=( )` instead:

```
// Common declarations
_LIT(KStringOne, "StringOne");
_LIT(KStringTwo, "StringTwo");

TBuf<128> buf1(KStringOne);
TBuf<128> buf2(KStringTwo);

// call TDesC::operator+=( )
buf1 += buf2;
```

You may also call the `TDes::Append( )` function:

```
buf1.Append(buf2);
```

In some (albeit rare) cases it may be appropriate to use the `TDesC::Insert( )` function. The next snippet employs the `HBufC` class for the sake of illustration. The code below ensures that the descriptor contains a trailing NULL character, which may be useful, for example, when your application needs to communicate with a server that expects C strings:

```
HBufC *pBuf = HBufC::NewLC(100);
// Append the string
pBuf->Des().Append(buf1);
// Add NULL character to the end
pBuf->Des().PtrZ();
// Increase pBuf's length to count the NULL terminator
TInt bufLen = pBuf->Des().Length();
pBuf->Des().SetLength(bufLen + 1);
// Call Insert() on the TDesC to keep the NULL
pBuf->Des().Insert(bufLen, buf2);
CleanupStack::PopAndDestroy(pBuf);
```

Please note that the `PtrZ( )` method doesn't change the length of the descriptor, so it is manually increased in the above sample.

Formatting is another alternative which we will investigate in more details in Section 4.1. The code snippet below shows how to use one of the numerous methods available in the `TDesC` class:

```
buf1 = KStringOne;
// Just do the formatting
buf1.AppendFormat(_L("%S"), &KStringTwo);
```

As an aside, the `_L( )` macro is deprecated for use in production code for efficiency reasons. I have not used it in many of the examples here but, in reality, you may prefer to use it rather than `_LIT( )` whenever it makes sense to do so, such as in the above example when we only need the literal once.

**What can go wrong:** The common reason for failing to concatenate two strings is that the destination descriptor (`buf1` in our snippets) is not large enough to accommodate the concatenated result. If this is the case, the code will panic with `USER 11`.

### 3.2 How to Format a Number into a String

**The problem:** You have an integer or a decimal value and want to convert it into a string representation.

**The solution:** TDes provides a number of conversion methods that you can use in any derived classes:

- TDes: : Num()
- TDes: : NumUC()
- TDes: : NumFixedWidth()
- TDes: : NumFixedWidthUC()
- TDes: : AppendNum()
- TDes: : AppendNumUC()
- TDes: : AppendNumFixedWidth()
- TDes: : AppendNumFixedWidthUC()

The above methods take either an integer or a decimal value as a parameter and allow you to convert it into the character representation according to a specified number system (for integers) or to the given format. The code snippet below illustrates a few possible calls:

```
TReal real Value = 16.0;
TUint value = 16;
TBuf<128> buf;

// Convert 64-bit unsigned integer
buf.NumUC(value);
// Convert 64-bit signed integer into its binary representation
buf.Num(value, EBinary);
// Convert 64-bit signed integer into its hexadecimal representation
buf.Num(value, EHex);

// Convert floating point number into character representation
TRealFormat fmt(KDefaultRealWidth, 2);
buf.Num(real Value, fmt);
```

Using Num-like methods is more efficient than Format() and so you should use them whenever possible.

### 3.3 How to Convert a Descriptor to a Number

**The problem:** You want to convert the character representation to the actual number.

**The solution:** This is an easy task on Symbian OS. Just look at the code snippet below:

```
_LIT(KNumber, "16");
TInt value = 0;
// Declare the variable
TLex lex(KNumber);
// Convert the descriptor into the integer number
TInt err = lex.Val(value);
if (err == KErrNone)
{
    console->Printf(_L("Use TLex to convert: Value = %i\n"), value);
}
```

```

    }
else
    {
        console->Printf(_L("Use TLex returned err: %i\n"), err);
    }
TReal32 realNum = 16.0;
err = TLex.Val(realNum, '.');

```

This sample illustrates the usage of the TLex class that will be covered in more detail in Section 5.5. The return value indicates the result of such a conversion, including the possible errors if the format of the character representation is not correct, such as containing non-numeric characters.

### 3.4 How to Externalize/Internalize a Descriptor

**The problem:** You want to load/save the descriptor from/to some binary representation.

**The solution:** You use either the operators << and >> or ReadL() and WriteL() methods of the stream classes – RDesReadStream, RFileReadStream, RMemReadStream and their RXWriteX counterparts. For the operators << and >> to be able to stream a class's data, that class has to implement two methods: ExternalizeL() and InternalizeL(). This allows you to load or save the exact binary layout of your class. Applying these general rules to the descriptors in particular, we may write the code similar to the next snippet:

```

void ExtIntSampleL()
{
    _LIT8(KDescExtInt, "This is a descriptor");

    RBuf8 rBufToExternalize;
    rBufToExternalize.CreateL(KDescExtInt(), 32);
    rBufToExternalize.CleanUpClosePushL();

    RBuf8 rBufToWriteInto;
    rBufToWriteInto.CreateL(64);
    rBufToWriteInto.CleanUpClosePushL();
    RDesWriteStream writeStream(rBufToWriteInto);
    writeStream.PushL();
    writeStream << rBufToExternalize;
    writeStream.Close();
    writeStream.Pop();

    RBuf8 rBufToInternalize;
    rBufToInternalize.CreateL(64);
    rBufToInternalize.CleanUpClosePushL();
    RDesReadStream readStream(rBufToWriteInto);
    readStream.PushL();
    readStream >> rBufToInternalize;
    readStream.Close();
    readStream.Pop();

    // Closes and destroys all RBuf variables
    CleanUpStack::PopAndDestroy(3);
}

```

**What can go wrong:** Operators << and >> may leave and so you should take this into account when developing your code.

### 3.5 How to Write Binary Data into a Descriptor or String Literal

**The problem:** You want to store binary data inside descriptors and literals.

**The solution:** You can use the `\xHH` sequences to put binary data into a `_LIT`:

```
//           A       l       e       x       e       y
_LIT(KBinaryStuff, "\x41\x00\x6C\x00\x65\x00\x78\x00\x65\x00\x79\x00");
```

This particular sequence defines the string 'Alexey' in Unicode. You can do the same trick with 8-bit literals. Since the descriptor always knows the size of its data, its behavior is quite similar to Basic strings and BSTR in the Windows world.

For regular descriptors, you may use the same approach with binary data using `Append()`:

```
TBuf8<32> buffer;
TUint8 data[6] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05};

// Put data into descriptor
buffer.Append(data, sizeof(data));

// Append the following byte values
buffer.Append(0x06);
buffer.Append(0x07);
buffer.Append(0x08);
```

**What can go wrong:** You would need to take care about byte-ordering issues when using the Unicode characters (namely `_LIT` or `_LIT16` but not `_LIT8`; the same is true for 16-bit descriptors), since `\xHH` in a Unicode string actually means `0x00HH`, and on Symbian OS will be stored as `0xHH, 0x00`.

## 4 More Complex Recipes

### 4.1 How to Format Strings

**The problem:** You have a number of different variables that you want to format and convert into a descriptor.

**The solution:** `TDes` provides several methods with the word 'Format' in their names. The following snippet demonstrates just a few of the possible formatting directives for various data types. You can find the full list of formatting syntax in Symbian Development Library documentation. So, here we go:

```
_LIT(KFmtBin, "Binary: %bb\n");
_LIT(KFmtOct, "Octal: %o\n");
_LIT(KFmtInt32, "Integer: %i\n");
_LIT(KFmtInt64, "64-bit integer: %Ld\n");
_LIT(KFmtReal, "Real: %f\n");
_LIT(KFmtUint, "Uint: %u\n");
_LIT(KFmtHex, "Hex: 0x%x\n");
_LIT(KFmtUint64, "Uint64: %Lu\n");
_LIT(KFmtCStr, "C-String: %s\n");
_LIT(KFmtDesc, "Descriptor: %S\n");

TBuf<128> buf1;
buf1 = KStringOne;
```

```

_LIT(KFormatSample, "FormatSample\n");
console->Write(KFormatSample);

TInt intValue = 16;
buf1.Format(KFmtBin, intValue);
console->Write(buf1);
buf1.Format(KFmtOct, intValue);
console->Write(buf1);
buf1.Format(KFmtInt32, intValue);
console->Write(buf1);
buf1.Format(KFmtInt64, (TInt64)intValue);
console->Write(buf1);
buf1.Format(KFmtHex, intValue);
console->Write(buf1);
buf1.Format(KFmtUint, -intValue);
console->Write(buf1);
buf1.Format(KFmtUint64, (TUint64)-intValue);
console->Write(buf1);

TBuf<128> buf2(KStringTwo);
buf1.Format(KFmtDesc, &buf2);
console->Write(buf1);
buf2.Append(0);
//const TUint16* pData = buf2.Ptr();
const TText* pData = buf2.Ptr();
buf1.Format(KFmtCStr, pData);
console->Write(buf1);

```

When you use the AppendFormat() function group, you can handle the possible buffer overflow by passing a pointer to the object of the class derived from TDesOverflow:

```

class TSampleOverflow : public TDesOverflow
{
    void Overflow(TDes& aDes);
};

void TSampleOverflow::Overflow(TDes& aDes)
{
    _LIT(KTextDescOverflow, "Got Descriptor overflow - max length = %d\n");
    console->Printf(KTextDescOverflow, aDes.MaxLength());
}

...
TSampleOverflow overflowHandler;
TBuf<9> buf1;
TBuf<128> buf2(KStringTwo);
// Here we reach maximum length of the descriptor
buf1.AppendFormat(KFmtDesc, &overflowHandler, &buf2);
// and here our overflow handler will be called
buf1.AppendFormat(KFmtDesc, &overflowHandler, &buf2);

```

**What can go wrong:** When you want to use descriptors in your formatting, you may wrongly use the upper case %S directive instead of %s and vice versa. '%S' is for a pointer to a descriptor while '%s' means 'char \*'. Also, forgetting the '&' sign before the descriptor variable means that you may waste a lot of time investigating why the code keeps crashing.

## 4.2 How to Convert a C String to a Descriptor (and vice versa)

**The problem:** You have a C string buffer and want to convert it into a descriptor (or vice versa).

**The solution:** The descriptors on Symbian OS usually don't contain a trailing NULL character (just like BSTR strings on Windows). Nevertheless, you can (i) easily put NULL at the end of the descriptor data and (ii) make a conversion back and forth between the descriptor representation and C strings.

```

_LIT(KFmtCStr, "C-String: %s\n");
_LIT(KFmtDesc, "Descriptor: %S\n");

TBuf<128> buf(KStringOne);
TUint cstrBuffer[100];
// Zero-fill the buffer
memset(cstrBuffer, 0, sizeof(cstrBuffer));
// Copy descriptor data to the C-string
memcpy(cstrBuffer, buf.Ptr(), buf.Size());
console->Printf(KFmtCStr, cstrBuffer);

// Make the opposite conversion
// PtrZ() method will add NULL character to the end of descriptor's data
const TText* text = buf.PtrZ();
console->Printf(KFmtCStr, text);

```

**What can go wrong:** If the descriptor's size is not large enough to accommodate an additional NULL character in the `TDes::PtrZ()` call, then you'll get a panic USER 23.

## 4.3 How to Convert an 8-bit String to a 16-bit String (and vice versa)

**The problem:** You have either an 8- or 16-bit descriptor and want to convert it to the desired character width.

**The solution:** The actual solution depends on your particular task. In its simplest form, all you might want to do is to have the raw conversion from, for example, ASCII representation to UNICODE. If that is the case, then all you need to do is to execute the following lines of the code:

```

_LIT(K8bitDesc, "8-bit");
TBuf8<128> buf1(K8bitDesc);
// Convert 8-bit descriptor to 16-bit and print it
TBuf16<128> buf2;
buf2.Copy(buf1);
console->Write(buf2);

```

As you can see, `TDes::Copy()` method does the job for you. Such conversion can be quite useful in many cases, for instance, when you work with networking code where the data tends to be of an 8-bit nature.

`Collapse()` and `Expand()` can also be used for raw conversions between 8- and 16-bit descriptors:

```

_LIT(K16bitDesc, "16-bit");
TBuf<128> buf1(K16bitDesc);
TPtr8 narrowPtr = buf1.Collapse();
TPtr widePtr = narrowPtr.Expand();

```

For more complicated conversions, especially if you need to work with national characters or UTF encodings, you need to use the `CnvUtfConverter` class:

```

HBufC8* pHeapUTF = CnvUtfConverter::ConvertFromUnicodeToUtf8L(buf2);
CleanupStack::PushL(pHeapUTF);
HBufC* pHeapUCS2 = CnvUtfConverter::ConvertToUnicodeFromUtf8L(*pHeapUTF);
CleanupStack::PushL(pHeapUCS2);
console->Write(*pHeapUCS2);
CleanupStack::PopAndDestroy(2);

```

For the latter snippet, you should link your code with `charconv.lib`.

**What can go wrong:** If you work with national character sets then some letters may be converted incorrectly.

## 4.4 How to Pass Descriptors as Function Arguments

**The problem:** You want to use a descriptor as a parameter for the function.

**The solution:** The standard technique is to use `TDesC` and `TDes` classes for that purpose. By doing this, you don't restrict the caller of the function to a specific descriptor type:

```

void FuncWithReadOnlyParameter(const TDesC& aConstData);
void FuncWithModifiableParameter(TDes& aData);

```

Using the above definitions, you could pass, for instance, `TPtr` or `RBuf` or any other variables of the numerous descriptor types derived from `TDesC` and `TDes`. Obviously this narrows the available operations with such parameters to those functions that are in the scope of the base class. In case you deliberately intend to work with `TPtr` rather than with `TDes`, for example, only do so if you have a good reason. But perhaps you could give it a second thought to ensure that you really need to. Base descriptor classes already offer many common operations and so specialized ones rarely make any difference.

Now let us have a closer look at the parameters in the above functions. You might think that it is peculiar to use 'const' for the non-modifiable descriptor in the first method. Well, that is true, it is non-modifiable according to the Symbian OS meaning of the term,<sup>1</sup> but you may still change its data using `TPtr`:

```

TPtr hackedTDesC(aConstData.Des());
hackedTDesC.Set(anotherTPtr);

```

Using 'const' enforces the C++ syntax and so you may prevent such crafty operations even at compile time. It also means that you can pass in literals using `_L()`, which you can't do if the parameter wasn't const, as this is illegal in C++.

**What can go wrong:** The obvious thing to do in a resource-restricted environment is to pass potentially large blocks of data 'by reference,' so that you don't fill the stack, as you would with the 'by-value' method (hence '&' in the parameters definition in above sample). Leaving aside the possible misuse of the parameters (we can be very creative in this matter), then a descriptor's overflow is a common problem that you may experience in this area. Remember that memory management is the developer's doom!

---

<sup>1</sup> The data contained in a non-modifiable buffer descriptor can be accessed, but not changed, through this descriptor. The data can, however, be changed using a `TPtr` descriptor, or completely replaced using the assignment operators.

Also, never attempt to instantiate base descriptor classes. Although SDK documentation describes them as 'abstract,' they are not from the C++ point of view. Thus you will be able to compile your code, but it won't work to your expectations.

## 4.5 How to Pass and Return Modifiable Descriptors

**The problem:** You want to communicate modifiable descriptors to and from functions.

**The solution:** As you have seen in the previous section, all you need to do to make the function's parameter modifiable is to declare it as TDes:

```
void SomeFunction(TDes& aData);
```

Inside the function you can both read from and write to the parameter, assuming it has enough space for the latter operation.

Now let us see how to return modifiable descriptors from the function. The following code snippet demonstrates one possible case:

```
TPtr TDesCDerivedClass::RightL(TInt aLength) const
{
    if (aLength < 0)
    {
        User::Leave(KErrArgument);
    }
    TInt len=Length();
    if (aLength>len)
    {
        aLength=len;
    }
    return(TPtr((TUint16*)Ptr()+len-aLength, aLength, aLength));
}
```

Just like in Standard C++, the return values are pointer-like, that is, you will use TPtr, HBufC\* or other classes that have the data that does not reside on the stack.

**What can go wrong:** The craftiest mistake is that you may forget to declare the parameters as reference types.

## 5 Advanced Recipes

### 5.1 How to Convert from One Descriptor Type to Another

**The problem:** You have a descriptor of one type and want to convert it to another one.

**The solution:** Look at the example code below for numerous conversions between descriptor types. You will see a combination of constructors, assignment operators and Des() and Ptr() methods:

```
_LIT(KTestBuf, "Buffer");
TBufC<16> bufC1(KTestBuf);
TBuf<16> buf1(bufC1.Des());

// TBuf -> TBufC
TBuf<16> buf2(KTestBuf);
TBufC<16> bufC2(buf2);
```

```

// TBuf/TBufC -> TPtr
TPtr ptr1(0, 0);
ptr1.Set((TUint16*)(buf2.Ptr()), buf2.Length(), 32);
TPtr ptr2(bufC2.Des());

// TPtr -> TBuf/TBufC
TBuf<16> buf3 = ptr1;
TBufC<16> bufC3 = ptr2;

// TBuf/TBufC -> TPtrC
TPtrC ptrC1(ptr1);
TPtrC ptrC2(buf2);
TPtrC ptrC3(bufC2);

// All above -> HBufC
HBufC* pHeap1 = ptrC1.AllLocLC();
HBufC* pHeap2 = ptr1.AllLocLC();
HBufC* pHeap3 = bufC1.AllLocLC();
HBufC* pHeap4 = buf1.AllLocLC();

// or using the assignment operator
HBufC* pHeap5 = HBufC::NewLC(32);
*pHeap5 = ptr1;

// or via Copy method
HBufC* pHeap6 = HBufC::NewMaxLC(32);
pHeap6->Des().Copy(ptr2);

// Make a cleanup
CleanupStack::PopAndDestroy(6);

```

**What can go wrong:** Base and derived descriptor classes have `Ptr()` and `Des()` functions which may be quite confusing as to which to use and when. As a result, you may have some compilation errors.

## 5.2 How to Use RBuf when a Modifiable Dynamic Descriptor is Needed

**The problem:** You want to use a resizable descriptor.

**The solution:** The `RBuf` class combines the behavior of the pointer descriptor classes – `TPtr` and `HBufC` – and is more convenient to use. `RBuf` may own the memory, as in the following snippet:

```

// Create RBuf
RBuf rBuf;
rBuf.CreateL(6);
// Put to the cleanup stack
rBuf.CleanupClosePushL();
// Do something: copy, resize and append
_LIT(KHello, "Hello ");
_LIT(KWorld, "World!");
rBuf.Copy(KHello());
rBuf.ReAlloc(12);
rBuf.Append(KWorld);
// Close and destroy

```

```

CleanupStack::PopAndDestroy();

// or point to some other buffer:

// RBuf::Assign
HBufC* hBuf = KHello().AllocL();
rBuf.Assign(hBuf);
// Do something and then cleanup
...

```

You can find more information about the RBuf class in the article, 'An Introduction to RBuf,' which can be found at [developer.symbian.com/main/documentation/symbian\\_cpp/symbian\\_cpp/index.jsp](http://developer.symbian.com/main/documentation/symbian_cpp/symbian_cpp/index.jsp).

**What can go wrong:** As with all other descriptor types, the main enemy is the memory management. You might overflow the descriptor, or the buffer it points to might be accidentally released at the moment your code wants to deal with it. Another common problem is that you should remember to call RBuf::Close() and release the memory associated with the RBuf object. RBuf::CleanupClosePushL() partially helps you here, but be careful when you assign a new string to an existing RBuf variable or change the ownership of the memory chunks.

### 5.3 How to Find and Match Descriptors

**The problem:** You want to search given strings within descriptors.

**The solution:** Various forms of Find(), Match() and Locate() functions should fulfill any basic searches you may need to perform. In addition to the expected functionality according to their name, some of these functions use techniques known as 'folding' and 'collation.'<sup>2</sup> The sample below illustrates just one functional family – Match() and MatchF():

```

//
// Match strings
//
_LIT(KTxtMatchstr1, "*World*");
_LIT(KTxtMatchstr2, "*W?rld*");
_LIT(KTxtMatchstr3, "Wor*");
_LIT(KTxtMatchstr4, "Hello");
_LIT(KTxtMatchstr5, "*W*");
_LIT(KTxtMatchstr6, "hello*");
_LIT(KTxtMatchstr7, "*");

_LIT(KMatchStr, "Match %S\n");
_LIT(KTxtNotFound, "Not Found");
_LIT(KTxtFound, "Found");
_LIT(KFormatStr, "%S %S (idx = %d)\n");

_LIT(KTxtHelloWorld, "Hello World!");

const TBufC<16> bufc(KTxtHelloWorld);
TInt index;

```

---

<sup>2</sup> 'Folding' is a relatively simple way of normalizing text for comparison by removing case distinctions, converting accented characters to characters without accents, etc. 'Collation' is a much better and more powerful way to compare strings, and each locale will usually have a standard set of collation rules.

```

TInt pos;
TPtrC genptr;
TBufC<8> matchstr[7] = {
    *KTxMatchstr1, // "*World*"
    *KTxMatchstr2, // "*W?rld*"
    *KTxMatchstr3, // "Wor*"
    *KTxMatchstr4, // "Hello"
    *KTxMatchstr5, // "*W*"
    *KTxMatchstr6, // "hello*"
    *KTxMatchstr7 // "*"
};

// search using Match()
for (index = 0 ; index < 7; index++)
{
    pos = bufc.Match(matchstr[index]);

    if (pos < 0)
        genptr.Set(KTxNotFound);
    else
        genptr.Set(KTxFound);

    console->Printf(KFormatStr, &genptr, &matchstr[index], pos);
}

// search using MatchF(): 6th string is matched in this case
for (index = 0 ; index < 7; index++)
{
    pos = bufc.MatchF(matchstr[index]);

    if (pos < 0)
        genptr.Set(KTxNotFound);
    else
        genptr.Set(KTxFound);

    console->Printf(KFormatStr, &genptr, &matchstr[index], pos);
}

```

**What can go wrong:** You cannot match '?' and '\*' characters as there is no escape symbol for Match() and MatchF() functions, but MatchC() does not have such limitations.

## 5.4 How to Compare Descriptors

**The problem:** You want to compare two descriptors.

**The solution:** TDesC class has a few comparison methods that you can employ for textual and binary data. The sample below demonstrates several possible scenarios:

```

void CompareSample()
{
    // Compare() & CompareF() can compare any kind of data.
    // For binary data just use Compare().
    // For text use Compare(), CompareF() or CompareC()
}

```



## 5.5 How to Use TLex

**The problem:** You have a descriptor and want to perform lexical parsing of its data.

**The solution:** Parsing the string-like data is quite a common task. For instance, your application may process some commands or work with GPS sentences. In such cases, the TLex class helps you enormously. The next snippet performs simple command parsing:

```
// This function parses the incoming string
// and looks for 'exit' or 'help' commands
TInt ParseCommand(const TDesC& aCommand)
{
    TLex lex(aCommand);
    TInt res=KErrNone;
    TBool help=EFalse;
    TPtrC ptr;
    for(ptr.Set(lex.NextToken()); ptr.Length(); ptr.Set(lex.NextToken()))
    {
        if(!ptr.CompareF(_L("q")) || !ptr.CompareF(_L("quit"))
            || !ptr.CompareF(_L("exit")))
        {
            return KErrEof;
        }
        if(!ptr.CompareF(_L("help")))
        {
            help=ETrue;
        }
        else if(ptr.Length()==2)
        {
            if(!ptr.CompareF(_L("-H")))
            {
                help=ETrue;
            }
            else
            {
                // Handle the possible commands here as needed
            }
        }
    }
}
```

The power of the TLex class is better shown in the more complex analysis, for instance, when you parse GPS data:

```
// Parse GGA sentences like that below:
// $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
void GpsParseGGASentence(const TDesC &aNmea)
{
    TReal latitude, longitude;

    _LIT(KSpace, " ");
    RBuf tmpBuf;
    if (tmpBuf.Create(aNmea) != KErrNone)
        return;

    // Replace all commas to utilize TLex::NextToken() afterwards
```

```

TInt commaPos = -1;
while ((commaPos = tmpBuf.Locate(', ')) > KErrNotFound)
{
    tmpBuf.Replace(commaPos, 1, KSpace);
}
TLex nmeaSentence(tmpBuf);
// Here we keep tokens in variables just for debugging purposes
TPtrC firstToken = nmeaSentence.NextToken();
TPtrC secondToken = nmeaSentence.NextToken();
TPtrC thirdToken = nmeaSentence.NextToken();
TLex tokenLat(thirdToken);
tokenLat.Val(latitude);
TPtrC auxLat = nmeaSentence.NextToken();
if (auxLat.Compare(_L("S")) == 0)
{
    latitude *= -1;
}

TLex tokenLon(nmeaSentence.NextToken());
tokenLon.Val(longitude);
TPtrC auxLon = nmeaSentence.NextToken();
if (auxLon.Compare(_L("W")) == 0)
{
    longitude *= -1;
}
}

```

**What can go wrong:** The TLex class uses the space character as a token separator and so if your data contains spaces then you need to handle it properly. In general, you may be a bit confused dealing with marked and current parsing positions and so forth.

## 6 Conclusion

In this article we briefly discussed some descriptor operations leaving the others for your own experiments. I hope this article answered many of your 'how-to' questions and made you more descriptor-friendly. We obviously could not cover all of the possible topics in one go and so you are invited to refer to the Symbian Development Library documentation for more information and sample code.

The sample project that accompanies this article can be found at:

- [developer.symbian.com/main/downloads/papers/Descriptors/Descriptors.zip](http://developer.symbian.com/main/downloads/papers/Descriptors/Descriptors.zip).

Additional useful information about the descriptors can be found here:

- [descriptors.blogspot.com](http://descriptors.blogspot.com)
- [www.forum.nokia.com/info/sw.nokia.com/id/3397e4ea-c067-4a4a-bf16-9747e1968b3e/S60\\_Platform\\_Descriptor\\_Example\\_v2\\_0\\_en.zip.html](http://www.forum.nokia.com/info/sw.nokia.com/id/3397e4ea-c067-4a4a-bf16-9747e1968b3e/S60_Platform_Descriptor_Example_v2_0_en.zip.html).

Further information can also be found in Symbian Press books, details of which can found here:

- [developer.symbian.com/books](http://developer.symbian.com/books).

## 7 Author Profile



Alexey started to play with mainframes at the end of the 1980s, using Pascal and REXX, but soon switched to C/C++ and Java on different platforms before moving into mobile technologies. After working for almost a decade as a team leader and architect on Windows Mobile, he decided to join the Symbian Core Development team, originally working on Security and later on USB.

Alexey holds an MSc in Applied Mathematics and Physics from the Moscow Institute of Physics and Technology. He is an Accredited Symbian Developer and regular author at [www.developer.com](http://www.developer.com). He recently contributed several chapters to the popular Symbian Press book, **Quick Recipes on Symbian OS** (more information for this can be found at [developer.symbian.com/quick](http://developer.symbian.com/quick)).