

A Symbian C++ Primer for BREW Developers

Roy Ben Hayun

Reviewed by Raghavendra Ghatage and Colin Ward

Published by the Symbian Developer Network

Version: 1.0 – March 2008

1	INTRODUCTION	2
1.1	INTENDED AUDIENCE	2
1.2	HOW TO READ THIS ARTICLE	2
2	C PROGRAMMING	2
2.1	AEESTDLIB.....	3
2.2	IAPPLET, IMODULE, AEEAPPGEN.C AND AEEMODGEN.C	3
2.3	BREW INTERFACES	4
2.4	BUILT-IN DATA TYPES.....	5
2.5	NAMING CONVENTIONS	6
2.6	WIDE AECHAR	6
2.7	COOPERATIVE NON-PREEMPTIVE PROGRAMMING.....	7
2.8	AEECALLBACKS AND EVENT-DRIVEN MODEL	7
2.9	ERROR HANDLING	8
2.10	BREW INTERFACES, DEMONSTRATED BY COMMON USE CASES	10
2.11	APPLICATION PRIVILEGES	14
3	THE DISTRIBUTION SYSTEM	14
4	SDK, SIMULATOR AND VC++.....	15
5	FORUMS AND ONLINE RESOURCES	15
6	CONCLUSION	16
7	WHAT TO DO NEXT	16

1 Introduction

A native execution environment can be defined as a software abstraction on top of the hardware chip. In telecomms, the hardware is combined with a software-based, native runtime environment to become a mobile device.

In this article we shall discuss some of the similarities between the native environments of two major mobile devices - BREW and Symbian OS. Both of these provide a proven solution in the telecomms industry, acting as native environments that provide higher level applications with native services.

The article is split into several sections, each one discussing aspects mutual to both BREW and Symbian OS.

1.1 *Intended audience*

This article is not meant to give a generic introduction to Symbian OS in the standard approach and so its intended audience is mostly C and C++ developers who are familiar with BREW 3.x and would like an introduction to Symbian OS 9.x by exploring some of the similarities between the two environments.

This article is preparatory reading material for BREW-savvy developers who want to use their skills and knowledge to learn more about Symbian OS and possibly to develop their first Symbian C++ application. Some of the topics are demonstrated through code samples in C and C++ to convey the concepts involved and to provide a more 'hands-on' feeling.

However, please note that this article is not meant to be a porting guide, and porting to Symbian OS from BREW is not part of the discussion.

1.2 *How to read this article*

To build understanding of a new concept in the shortest amount of time, it is convenient to refer to some of the elements shared in another, already familiar concept.

Here is an illustrative example - a person who can speak fluent French would like to learn a similar Latin language, for example, Italian. It is likely that his first questions would be, "How do you say ... in Italian?" After a number of such "how do you say...?" questions, he still cannot speak Italian but he will already have an intuitive feel for the relationship between the two Latin languages. Those questions will give him a taster, after which he can begin some more structured learning.

The article exercises the above approach by choosing several BREW concepts and relating them to their implementation in Symbian OS (i.e., "how do you say ... in the language of Symbian OS?"). Therefore, the point of view chosen for this article is that of a C or C++ developer with sufficient familiarity with mobile development on the BREW 3.x platform. We take the concepts he or she is familiar with, and draw on those skills and knowledge to gain a glimpse into development for Symbian OS.

2 C programming

The lingua franca for development on the BREW platform is C. For Symbian OS it is C++ with a number of "extensions" – often referred to as "Symbian C++".

For a C or C++ developer familiar with BREW, a quick look at some Symbian C++ code could be sufficient to understand what it does in general. However, apart from differences in the language

syntax there are various Symbian OS platform-specific idioms internalized in the code that you need to be aware of. Familiarity with these will also give a deeper understanding of the implementation details and of how the code is executed overall.

These Symbian OS platform idioms are not deviations from the ANSI C++ standard - most of them are only guidelines - and not adhering to them does not necessarily result in build breaks (e.g., compilation errors); however, it may have consequences in other areas that are not necessarily technical (e.g., portability, readability of code or potential for memory leaks). In the following sections we refer to some of these idioms in more detail.

A large number of articles about Symbian C++ are available on the SDN website (developer.symbian.com/oslibrary).

2.1 AEEStdlib

BREW has an implementation of the C Standard Library, known as AEEStdlib. For example, consider the following code, which allocates an AECHAR array and uses the AEEStdlib, MALLOC and FREEIF macros:

```
pzFileNameBuf = (AECHAR*) MALLOC( MAX_FILE_NAME * sizeof( AECHAR ) );
if( !pzFileNameBuf )
{
    return ENOMEMORY;
}
...
FREEIF( pzFileNameBuf );
```

Symbian OS also has an implementation of the C Standard Library, known as STDLIB, which it provides along with the associated POSIX system APIs, in order to support both pure C programs and mixed C/C++ programs. The STDLIB was implemented to address the porting requirements of C programs for Symbian OS. However, the STDLIB was not designed to pass ANSI or POSIX conformance tests, or to be a C API replacement for Symbian OS.

In 2007, Symbian introduced P.I.P.S. (**P.I.P.S. Is POSIX on Symbian OS**). It supplies a framework of POSIX C APIs for use by both C and C++ programmers, which are packaged into industry standard libraries (`Libc`, `Libpthread`, `Libm`, and `Libdl`) and are tightly integrated with Symbian OS for optimized performance and memory usage.

P.I.P.S. improves developer productivity and it helps to broaden the development community for Symbian OS: by providing standard POSIX C APIs on Symbian OS, C programmers can migrate existing middleware and applications, either commercial or open source, to Symbian OS more easily.

Another set of C APIs available on Nokia S60 is Open C, a superset of P.I.P.S., which includes industry-standard POSIX and middleware C libraries for Nokia S60 devices. More information about Open C is available from www.forum.nokia.com/openc and you should also refer to developer.symbian.com for more information about Standard C APIs for Symbian OS.

2.2 IApplet, IModule, AEEAppGen.c and AEEModGen.c

A BREW applet is an implementation of the IApplet interface which handles events in the Application Execution Environment (AEE) in response to events generated by the system. The two files that are part of the BREW SDK, **AEEModGen.c** and **AEEAppGen.c**, provide a ready-made implementation glue to bind your application to the AEE. **AEEModGen.c** handles the module, which is essentially a container for a few applets. The **AEEAppGen.c** handles the applet, which defines the required functions for the abstraction of the application execution unit.

In this way, BREW provides a straightforward approach to application implementation. During runtime, when the user launches an application the AEE creates your module which, in turn, instantiates your application by calling the `AEECL_SCreateInstance()` function that each `IApplet` implementation must provide.

On Symbian OS all GUI-based applications use the Symbian OS application framework architecture, APPARC, and must provide a concrete implementation of several base classes provided by the Symbian licensee's UI platform, themselves derived from APPARC base classes (e.g., AVKON for Nokia's S60 UI platform or Qikon for UIQ).

Let us now briefly discuss the framework architecture of the applications which are compatible with Symbian OS. Most applications consist of the following four abstractions:

- an application
- a document
- an appUI
- a view.

The "application" is the abstraction of an application working within the framework. At application initialization time, it creates a "document", which is responsible for the application data as well as for the management of resources such as files and databases. The document creates the "appUI", which is not actually a graphical entity but the owner of the application graphical components and the different application views.

When the user launches an application, the application framework calls the application's entry point to launch the application.

The application framework works to a large extent in the same manner across the various UI platforms available for Symbian OS (e.g., Nokia S60 and UIQ). However, without giving the exact names of the Symbian OS application framework classes or the names of the derived classes in each OEM, we should note that there might be some minor adjustments between different UI platforms.

For more information on the Symbian OS application framework, and the derived classes per OEM, please refer to the OEM SDK documentation at developer.symbian.com/main/tools/sdks.

2.3 BREW interfaces

The concept of BREW interfaces is equivalent to the concept of classes in many object-oriented languages such as Java and C++, which define a set of member functions.

In Symbian OS a logical group of functions is encapsulated in a single object-oriented class or in a set of classes, which all belong to the relevant Symbian OS subsystem (e.g., networking or multimedia). In BREW, however, all of these features are simulated by organizing the C APIs into interfaces, each of which governs a functionality area, and by providing BREW macros to access their member functions.

The following code illustrates the instantiation of a BREW interface:

```
IShell * shell;
AEECLSID clsid;
void** newInterface;
int err;
...
err = ISHELL_CreateInstance(shell, clsid, newInterface);
...
```

The equivalent to this code in Symbian C++ would be to construct a new object using a two phase construction idiom, implemented with a static class function, `NewL()`, that returns a pointer to a new instance:

```
CMyClass* myClass = MyClass::NewL(value);
```

The list of supported BREW interfaces is not directly equivalent to the list of classes in Symbian OS, but is closer to the combination of a Symbian OS subsystem and a relevant Symbian C++ class.

For example, the `ISockPort` interface that was introduced in BREW 3.x governs the functionality for network-type communication. In Symbian OS, network communication is governed by the Symbian OS Comms subsystem and `ESock`. Inside the Comms subsystem, the `RSocket` class acts as an endpoint for network-type communication, and provides functions for socket creation, reading, writing and more. A more elaborate explanation of `ESock` and a code example for a simple TCP client is provided in the following sections.

Typically, the header file included refers to the subsystem, and the instantiated object would be an instance of the relevant encapsulating class.

For more information on the various subsystems and classes in Symbian OS, please refer to the Symbian OS guide and Symbian OS reference documentation at www.symbian.com/developer/techlib/v9.3docs/doc_source/index.html.

2.4 Built-in data types

A BREW developer would use the built-in data types, defined in **AEEComdef.h**, throughout the code (e.g., `boolean`, `int16`, `uint32`, etc.).

In Symbian OS, there are primitive data types that should always be used in preference to their C++ built-in equivalents.

There are four general kinds of class, one of which is the value class, or built-in types that are given typedef names beginning with 'T' (e.g., `TInt`). Types that satisfy the criteria for the T class do not own any external resource, either directly (by pointer) or indirectly (by handle). As such, all built-in C++ types satisfy the criteria for the T class.

Below is a list of typedef names for the standard built-in data types:

```
typedef signed char TInt8;
typedef unsigned char TInt8;
typedef short int TInt16;
typedef unsigned short int TInt16;
typedef long int TInt32;
typedef unsigned long int TInt32;
typedef signed int TInt;
typedef unsigned int TInt;
typedef float TReal32;
typedef double TReal64;
typedef double TReal;
typedef unsigned char TText8;
typedef unsigned short int TText16;
typedef int TBool;
```

Although not mandatory, it is strongly recommended that the typedef names are used instead of the built-in C++ data types.

2.5 Naming conventions

In Symbian C++ there is heavy usage of naming conventions that are unique to Symbian OS.

Consider two lines of code that were used in the previous sections:

```
CMyClass* myClass = CMyClass::NewL(value);
typedef unsigned short int TInt16;
```

There is a prefix 'C' in the class name, trailing 'L' in the class `NewL()` method and a prefix 'T' in the typedef.

There are many more examples we could give but we will not stop here to explain these naming conventions as this will divert us from the focus of the discussion. However, naming conventions are fundamental to Symbian OS programming and care must be taken when writing or reading code (the *Coding Standards* booklet from Symbian Press provides a full explanation of these conventions, and it can be found at developer.symbian.com/booklets). A specific example for the importance of Symbian OS naming conventions (related to the trailing 'L' in the `NewL()` method) is mentioned in section 2.9 which discusses error handling.

2.6 Wide AECHAR

In BREW, a string can be declared in standard C or you can use `AECHAR`, which is a typedef for `uint16` (see `AEE.h`). For example, the following code allocates two strings on the heap and compares them using the `AEEStdlib` string comparison macro:

```
buf1 = (AECHAR*) MALLOC( ( MAX_SIZE + 1 ) * sizeof( AECHAR ) );
buf2 = (AECHAR*) MALLOC( ( MAX_SIZE + 1 ) * sizeof( AECHAR ) );
...
int same = WSTRCMP(buf1, buf2)
```

In Symbian OS, strings and binary data buffers are represented by descriptors, which make it easy to store, manage and describe string-based data. As descriptors may contain any data, they are commonly used to store general buffers as well.

The name 'descriptors' means that they are self-describing and this is what makes them very different from standard C strings. Unlike C strings that do not know their length and have the NULL character to indicate the end of the string, descriptors do know the length of the data stored in the buffer, and they also identify the underlying memory layout of the data (e.g., stack or heap).

They store text data as either ASCII or Unicode, they are not NULL terminated and they are not difficult to code with, once you get used to thinking in this way.

Descriptors solve various problems posed by standard C strings, for example:

- When you use C strings from the heap or stack, you are required to store the length in an additional variable in their code.
- Standard C strings are unsafe and introduce a security risk which may cause an unknown behavior or a crash (i.e., due to buffer overflow), or which may even be exploited by malicious software.
- With standard C strings, you must prevent buffer limits overflow. However, descriptors can tell the data length and the maximum length of the stored data, and can provide inherited support for the maximum length.

Please note that you can still use standard C arrays to store data and can manipulate them using the C string library, but all Symbian C++ APIs use descriptors for input and output parameters, and so the conversion would be required repeatedly.

Here is some example code to illustrate the allocation of memory on the heap, which is similar to the previous BREW code sample:

```
HBufC* buf1 = HBufC::NewL(MAX_SIZE);
HBufC* buf2 = HBufC::NewL(MAX_SIZE);
...
TInt same = buf1->Compare(*buf2);
```

To further illustrate the use of descriptors, here are three cases where the same functionality is provided by using both AEEStdlib C string manipulation and the descriptor method:

- As shown in the above example, comparing C strings lexicographically using AEEStdlib STRCMP or WSTRCMP is the equivalent of using TDes::Compare().
- Writing formatted data to a C string using AEEStdlib functions SPRINTF or SNPRINTF is equivalent to using TDes::Format().
- Appending a string to another C string using AEEStdlib functions STRCAT WSTRCAT is the equivalent of using TDes::Append().

For more information about descriptors, you can refer to a range of Symbian OS programming books: they always contain a section about descriptors because they are fundamental to Symbian OS programming and so it is so essential to understand them well. In particular, *Symbian OS C++ for Mobile Phones Volume 3*, by Harrison and Shackman, provides a thorough explanation of descriptors. See developer.symbian.com/books for further details of this and the other books available in the Symbian Press range.

2.7 Cooperative non-preemptive programming

BREW programming uses non-preemptive multitasking in which it is essential not to use a busy loop and poll for data, otherwise the underlying OS watchdog that checks the health of the running process will shut down the process or restart the device. It is your responsibility to yield control to the system during long-running operations.

Symbian OS has support for preemptive multitasking using the RThread class which contains a handle to a thread that is itself a kernel object. In this way, you can start a long-running process without needing to break it into short-running segments or to periodically relinquish control. The support for preemptive multitasking raises the issue of synchronization, which is handled in Symbian OS with the RSemaphore and RMutex kernel objects. (Please note that mutexes and semaphores are not Symbian C++ idioms but are standard operating system constructs.)

However, although the RThread option is there to support preemptive multitasking, switching between threads is still considered resource-consuming for mobile devices. Instead, programmers are encouraged to use cooperative multitasking instead, using active objects (another Symbian C++ idiom which we will cover in more detail in the following section).

For more information on the options for preemptive and cooperative multitasking please refer to the Symbian Developer Network technical papers and Symbian Press books at developer.symbian.com/main/oslibrary and developer.symbian.com/books respectively.

2.8 AEECallbacks and event-driven model

BREW offers an asynchronous, event-driven model. AEECallbacks are a data structure used for scheduling operations that notify the client of an event via a callback function, and which encapsulate the most commonly needed support, such as which client to call when the event occurs and with what parameter, etc.

The following code sample makes use of the `AEECallback m_cbReadCallback` variable:

```
// no data is available for reading
if (AEEPORT_WAIT == ret)
{
    // TryRead() will be called again when the read operation can progress
    ISOCKETPORT_ReadableEx(m_pISocketPort, m_cbReadCallback, TryRead, NULL);
    return;
}
```

In Symbian OS, the idiom used for receiving notifications of asynchronous events is known as the active object. At the core of the active object idiom are the `CActive` and `CActiveScheduler` classes.

The `CActive` class encapsulates both the issuing of asynchronous requests, as well as their handling once they are completed. `CActiveScheduler` controls the sequence in which active object request completion events are handled.

This is a simplified explanation of how the active objects and active scheduler work together:

- The client passes the active object's request status to the service provider, which changes the value of the request status when the event is ready to be consumed.
- The active scheduler, which is monitoring and waiting for value changes of request statuses, then invokes the active object's event-handling method (`CActive::RunL()`) which processes the event.

Although there is a lot more to it than this, this overview of how it works allows the similarities between BREW and Symbian OS event-handling idioms to become clear.

One implication of the different implementation approaches (C functions compared to an object-oriented approach), however, is that the callback context in the BREW macro `CALLBACK_Init(pcb, pcf, pcx)` can be encapsulated as data members in a `CActive`-derived class instead.

For example, in BREW programming, if a pointer to relevant data is needed when the event is to be consumed in a BREW callback method, it can be put as the `pcx` parameter in the `CALLBACK_Init` macro. In Symbian OS the relevant pointer can be a member in a `CActive`-derived class and, as such, it would be accessible to the handling active object when the `RunL()` method is invoked.

Although in BREW C callback functions are used with a passed context parameter, and in Symbian OS there is the `CActive` class to derive from, in essence the principle of cooperative multitasking works in a similar manner.

For more information about active objects, we recommend you refer to any book about Symbian OS development from the Symbian Press range (developer.symbian.com/books) or the paper on active objects that can be found on the Symbian Developer Network at developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf.

2.9 Error handling

In BREW, errors are handled in various ways:

- You are required to check the return value of BREW API methods that return an error code or a successful value and handle both cases.

- For other functions that return void, you can use the BREW API GetLastError() mechanism.
- In asynchronous operations, the error is notified in the callback function.

In Symbian OS, error codes from synchronous operations that return a value must be checked and handled in the same way. Error codes from asynchronous operations are handled when the relevant event-consuming active object's RunL() is invoked, at which time you must check the member request status value, and handle both successful and failed completion results.

Some commonly used BREW error codes and their equivalent Symbian OS error codes

BREW Error Code	Symbian OS Error Code
SUCCESS	KErrNone
EFAILED	KErrGeneral
ENOMEMORY	KErrNoMemory
EBADPARM	KErrArgument
EUNSUPPORTED	KErrNotSupported

There is an additional, fundamental mechanism for handling errors on Symbian OS, which uses an idiomatic approach to provide the automatic cleanup of resources after an unexpected error occurs. You can push resources onto a cleanup stack that takes responsibility for releasing them in case a “leave” happens. A leave is a lightweight exception-like mechanism: methods leave when they cannot handle an error and need to propagate it to code that can do so. When a leave occurs, the cleanup stack is invaluable for releasing allocated resources, although, alternatively, you can do this explicitly.

For example, SafeMethodL() illustrates how the cleanup stack can be used to avoid resource leaks when calling CBook::DoSomethingL() – which is identified as a method that may leave because it follows the Symbian OS naming convention of having a trailing L at the end of the function name.¹

```
void SafeMethodL()
{
    // Heap allocation
    CBook* book = new(ELeave) CBook();
    // Push book onto the cleanup stack
    CleanupStack::PushL(book);
    book->DoSomethingL(); // May leave
    // Pop from the cleanup stack
    CleanupStack::Pop(book);
    delete book;
}
```

A pointer to the CBook object is pushed onto the cleanup stack so that the heap memory the object occupies can be freed in the event that the DoSomethingL() method leaves. If DoSomethingL() does not leave and returns successfully, book is popped off the cleanup stack and destroyed normally.

¹ Note that the trailing L is only a matter of adhering to coding standards. There is nothing that guarantees that a method without a trailing L in its name would not throw leave. However, this is considered a serious violation of the coding standards that introduces a risk to the program robustness by not communicating the full method contract in code.

2.10 BREW interfaces, demonstrated by common use cases

Having become familiar with quite a few of the most fundamental Symbian C++ programming concepts, it is now time to demonstrate how common BREW use cases are coded in Symbian C++.

In this section, we refer to some BREW interfaces governing functionality for two common use cases, and then refer to the Symbian OS subsystem that governs this functionality, using some more elaborate code examples to illustrate the differences between the two implementation approaches and to gain an insight into a few more Symbian C++ concepts.

2.10.1 Example 1: answering incoming phone calls

A BREW telephony application that answers incoming calls usually performs the following steps:

1. Register a model listener with the ITelephone interface:

```
{
...
ITelephone_QueryInterface(pMe->pTelephone, AEELID_MODEL, (void**) &pModel);
nError = IModel_AddListenerEx(pModel, &pMe->phoneListener,
                             HandleTelephoneEvent, pMe);
...
}
```

2. Obtain access to ICall object when model listener is invoked:

```
void HandleTelephoneEvent(CApplet* pMe, ModelEvent* pEvent)
{
    AEETCallEvent * pCallEvent = (AEETCallEvent*) pEvent;
    if (pCallEvent && (AEET_EVENT_CALL_INCOM == pCallEvent->evCode))
    {
        AEETCallEventData * pTCallEventData =
            (AEETCallEventData *) &(pCallEvent->call);
        ICallMgr_GetCall(pMe->pCallMgr, pTCallEventData->cd, &pMe->pIncomingCall);
    }
}
```

3. Obtain call information identified by the ICall object using ICall_GetInfo():

```
{
    AEETCallInfo sTCallInfo;
    MEMSET(&sTCallInfo, 0, sizeof(sTCallInfo));
    nError = ICall_GetInfo(pCall, &sTCallInfo, sizeof(sTCallInfo));
}
```

4. Answer the call:

```
nError = ICall_Answer(pMe->pIncomingCall);
```

We will now discuss, in high level, the implementation of the same use case, of answering incoming calls, in Symbian C++:

1. Include the headers of the relevant Symbian OS telephony subsystem:

```
#include <e32base.h>
#include <Etel3rdParty.h>
```

2. Derive from CActive and define your own active object class:

```
class CClientApp : public CActive
{
```

3. Hold references to context objects (like pcx in your AEECall back):

```
private:
CTel ephony* iTel ephony;
CTel ephony: :TCa l l I d i Ca l l I d;

public:
CCl i entApp(CTel ephony* aTel ephony);
voi d AnswerCa l l ();

private:
/*
```

These are the pure, virtual methods from CActive which **must** be implemented by all active objects:

```
*/
voi d RunL();
voi d DoCancel ();
};

// CCl i entApp class Implementation
CCl i entApp: :CCl i entApp(CTel ephony* aTel ephony): CActi ve(EPri ori tyStandard),
iTel ephony(aTel ephony)
{
// default constructor
}
}
```

4. Provide a service API method to initiate the operation:

```
voi d CCl i entApp: : AnswerCa l l ()
{
iTel ephony->AnswerI ncomi ngCa l l (i Status, i Ca l l I d);
SetActi ve();
}
```

5. Handle the event, when RunL() is invoked by the active scheduler (like HandleEvent() in your BREW application):

```
voi d CCl i entApp: : RunL()
{
if(i Status==KErrNone)
{} // The call has been answered successfully;
// iCa l l I d contains the call's ID, needed when controlling the call.
}
```

6. Provide a cancel method (like CALLBACK_Cancel in your BREW application):

```
voi d CCl i entApp: : DoCancel ()
{
iTel ephony->Cancel Async(CTel ephony: : EAnswerI ncomi ngCa l l Cancel );
}
```

2.10.2 Example 2: a simple TCP client

A simple BREW TCP client application that reads and writes data over an ISockPort object usually performs the following steps:

1. Data initialization:

```
// initialize the addresses
pME->m_saSockAddr.wFamily = AEE_AF_INET;
pME->m_saSockAddr.inet.port = HTONS(SERVER_PORT);
INET_PTON(pMe->saSockAddr.wFamily, SERVER_ADDR, &(pMe->saSockAddr.inet.addr));
```

2. Creating and opening an I Socket:

```
// create the I Socket object.
ret = ISHELL_CreateInstance(pME->m_plShell, AEECLSID_SOCKET, (void**)&(pME->
m_plSocket));
...
// open the Socket.
ret = ISOCKET_OpenEx(pME->m_plSocket, AEE_AF_INET, AEE_SOCKET_STREAM, 0);
```

3. Establish a connection to the TCP server:

```
// connect to the distant server
ret = ISOCKET_Connect(pME->m_plSocket, &pME->m_saSockAddr);
if (AEEPORT_WAIT == ret)
{
    ISOCKET_WriteEx(pME->m_plSocket, &pME->m_cbWriteCallback,
                    CApp_TryConnect, pME);
    return;
}
```

4. Writing and reading data to and from the TCP server:

```
ret = ISOCKET_Write(pME->m_plSocket,
                    pME->m_caWriteBuffer + pME->m_nBytesWritten,
                    BUFFER_SIZE - pME->m_nBytesWritten);

// retry later
if (AEEPORT_WAIT == ret)
{
    ISOCKET_WriteEx(pME->m_plSocket, &pME->m_cbWriteCallback,
                    CApp_TryWrite, pME);
    return;
}
```

5. Closing the I Socket:

```
CALLBACK_Cancel(&pME->m_cbReadCallback);
CALLBACK_Cancel(&pME->m_cbWriteCallback);
IBASE_Release((IBase*)(pME->m_plSocket));
```

We will now discuss the implementation of the same use case in Symbian C++, which involves active objects and another Symbian OS architectural idiom - the client-server framework. (Please note that we will not discuss every implementation detail, so that we can give a high level overview of the process where possible.)

1. Derive from class CActive to create our own active object:

```
#include <e32base.h>
#include <in_sock.h>
#include <es_sock.h>

class CTCPConnector : public CActive
{
private:
    // these are some of the classes relevant to opening a TCP connection:
    TInt iState;
```

```

RSocket iSocket;
RSocketServ iSocketServer;
RHostResolver iResolver;
TInetAddress iAddress;

```

2. Define the service function:

```

void CTCPConnector::MakeOutgoingConnectionL(const TDesC& aHost, TInt aPort)
{
    ...
    iState = EGetByName;
    iResolver.GetByName( /* parameters required for resolving a host */);
    ...
}

```

3. Define a method to connect:

```

void CSEIConnector::ConnectSocketL(void)
{
    ...
    iSocketServer.Connect();
    ...
    iSocket.Open(iSocketServer, KAfInet, KSocketStream, KProtocolInetTcp);
    ...
    iSocket.Connect( /* parameters required connecting */);
    ...
    iSocket.Connect(iAddress, iStatus);
    iState = ESocketConnect;
    ...
}

```

4. Define RunL() to handles event notifications:

```

void CTCPConnector::RunL()
{
    TInt error = KErrNone;
    switch(iState)
    {
        case EGetByName:
        {
            ConnectSocketL();
            break;
        }
        case ESocketConnect:
        {
            ProcessRequestL();
            break;
        }
    }
}

```

(There are parts missing from this code but this is to help us focus on only a few specific events at this time.)

Note the code from the method `ConnectSocketL(void)`, which makes use of the `iSocketServer` and `iSocket` members:

```

iSocketServer.Connect();
...
iSocket.Open(iSocketServer, KAfInet, KSocketStream, KProtocolInetTcp);

```

The client-server in this sense is not to be confused with a TCP Client-Server as it is, in fact, an architectural idiom used by Symbian OS to manage and synchronize access to a shared resource.

An application uses the Client class to receive services from a Server class, and so, to receive TCP services from the OS, we create and open an RSocket Client class which receives those services from a RSocketServer Server class.

The use case would run as follows:

1. MakeOutgoingConnect() is invoked by the client.
2. MakeOutgoingConnect() initiates a DNS lookup.
3. When the host name is resolved, the CTCPConnector::RunL() is invoked.
4. In CTCPConnector::RunL(), we check the request status for success/error and the iState to know which stage of the process we have reached.
5. If our state is EGetByName, we call ConnectSocketL().
6. In ConnectSocketL(), we create the Client and Server classes, RSocket and RSocketServer respectively.
7. Invoke RSocket::Connect() to create the actual TCP connection.
8. When the connection succeeds or fails, RunL() is invoked again.

Although not every detail was covered in these two examples, hopefully they have given you a taste of Symbian OS programming and a more practical experience and understanding.

Other, more complete, examples are available from the Symbian Developer Network website at developer.symbian.com.

2.11 Application privileges

BREW security centers around the application which is authenticated, protected and granted privileges. The privileges that are contained in the application's MIF determine whether the module's classes are allowed to use certain BREW API functions.

Symbian OS v9 introduced capabilities and a Platform Security framework under which, to use protected APIs, an application has to be granted with platform security capabilities. On Symbian OS, protected APIs are those that allow sensitive operations, such as those that require access to end users' private data, the creation of billable events, access to the mobile phone network, and access to handset functions that can affect the normal behavior of the phone or will potentially impact the performance of other applications running on the phone.

As in BREW, the platform security capabilities are not programmed in the code but are defined in the project's meta files. You must specify the required capabilities in the Symbian OS build files and those are granted through the Symbian Signed program, a signing program similar in some ways to that which is offered by BREW, as the next section will explain.

3 The distribution system

The BREW distribution system allows network operators to manage BREW application downloads and provides a full management solution for the distribution and billing issues. Each application available for download has to be certified through the TRUE BREW scheme before it is deployed. Developers are issued a key by VeriSign with which to create an application with a ClassID and sign up the application, which needs to be signed again with a BREW signing key.

Symbian Signed is the mobile industry endorsed certification program that promotes best practice in designing applications to run on Symbian OS phones. Symbian Signed applications follow industry agreed quality guidelines and support network operator requirements for applications and content. Symbian Signed defines specific tests which you must comply with and requires that the application be tested to verify that the quality requirements are indeed met. The application certification process ensures that the application origin is known, that high quality standards are kept, and that an extra layer of security is provided to protect the end user from malicious software.

Users can download applications from any available source as Symbian OS was designed to be an open environment.

For more information on the Symbian Signed program, and the various options available for signing applications, please refer to www.symbiansigned.com.

4 SDK, simulator and VC++

BREW developers can download the BREW SDK, which contains documentation and development tools such as the simulator and the plug-in to MS Visual Studio.

The open Symbian OS UI Platforms (e.g., S60, UIQ) each host a developer portal from which you can download a Symbian OS SDK. More information about where to find the portals can be found at developer.symbian.com/main/getstarted/developer_community, and links to the SDKs can be found at developer.symbian.com/main/tools/sdks/. Each SDK contains a wealth of documentation, code examples and the Symbian OS emulator.

The IDE of choice for Symbian OS is Carbide.c++, which is a family of Eclipse-based development tools for Symbian OS on S60 and UIQ, and which can be extended with other Eclipse plug-ins and packages. Carbide.c++ focuses on three primary development areas: development tools for Symbian C++, development tools for Java, and user interface personalization and customization.

The emulator includes many features which enhance its usability, and provides a powerful development environment running under Windows. Most development effort is PC-based and usually only the final development stages require running and debugging on the target hardware, which is also supported.

5 Forums and online resources

BREW developers can post questions in the BREW forums and benefit from a large number of articles and presentations on the BREW home website (brew.qualcomm.com) where they can also download a BREW development kit.

Developers working on Symbian OS can post their questions in the various SDN developer forums (developer.symbian.com/forum) and in other forums hosted by Symbian OS licensees (such as those hosted by Forum Nokia at www.forum.nokia.com/main.html, and by UIQ at developer.uiq.com/forum/forumindex.jspa). Articles and presentation are available throughout the SDN on various topics related to Symbian OS development.

There are also a large number of valuable online resources such as newlc.com and allaboutsymbian.com, amongst others.

Symbian Press offers a large number of books covering areas including Symbian C++ programming, general topics relating to the Symbian OS architecture, specific Symbian OS components and more – information is available at developer.symbian.com/books.

6 Conclusion

In this article we discussed a set of concepts which are fundamental to BREW application development. Through these concepts we have tried to provide an insight into Symbian OS concepts.

We have covered the main programming language for Symbian OS, available C programming libraries, the basic execution unit - the application abstraction, class-based encapsulation, primitive data types, strings and buffers, multitasking and preemption, events-handling, error handling, security, distribution, development tools and knowledge bases.

Clearly this is not enough information and experience for programming your first Symbian application immediately, but hopefully we have managed to answer a few "how do you say ... in the language of Symbian OS?" questions.

7 What to do next

A C or C++ developer who is familiar with BREW mobile development can find out more about Symbian OS by reading the articles that are linked from this one and referring to the large number of online resources mentioned, as well as the books (developer.symbian.com/books) and booklets (developer.symbian.com/booklets) available from Symbian Press which are listed on the SDN website, at developer.symbian.com.

By downloading an SDK (from developer.symbian.com/main/tools/sdks/), you can start programming your first Symbian C++ application in the [Carbide.c++ IDE](http://www.forum.nokia.com/main/resources/tools_and_sdks/carbide/index.html) (available from www.forum.nokia.com/main/resources/tools_and_sdks/carbide/index.html), and can then run it on the Nokia S60 or UIQ emulator (and further information regarding these can be found at developer.symbian.com/main/tools/sdks/s60/ and developer.symbian.com/main/tools/sdks/uiq/ respectively).

It only remains for me to welcome you to the Symbian ecosystem!

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.