New IPC Mechanisms for Symbian OS

By John Pagonis, February 2005
Version 1.1

Intro

In the following we examine the new Inter-Process Communication mechanisms available to applications and servers in Symbian OS. Understanding when and how these are required is essential in properly architecting components and making optimum use of the OS capabilities. Each of these mechanisms has its own strengths and limitations, so utilizing the right one will result in cleaner, more efficient programs that are easier to understand and maintain.

Evolution of Symbian OS architecture

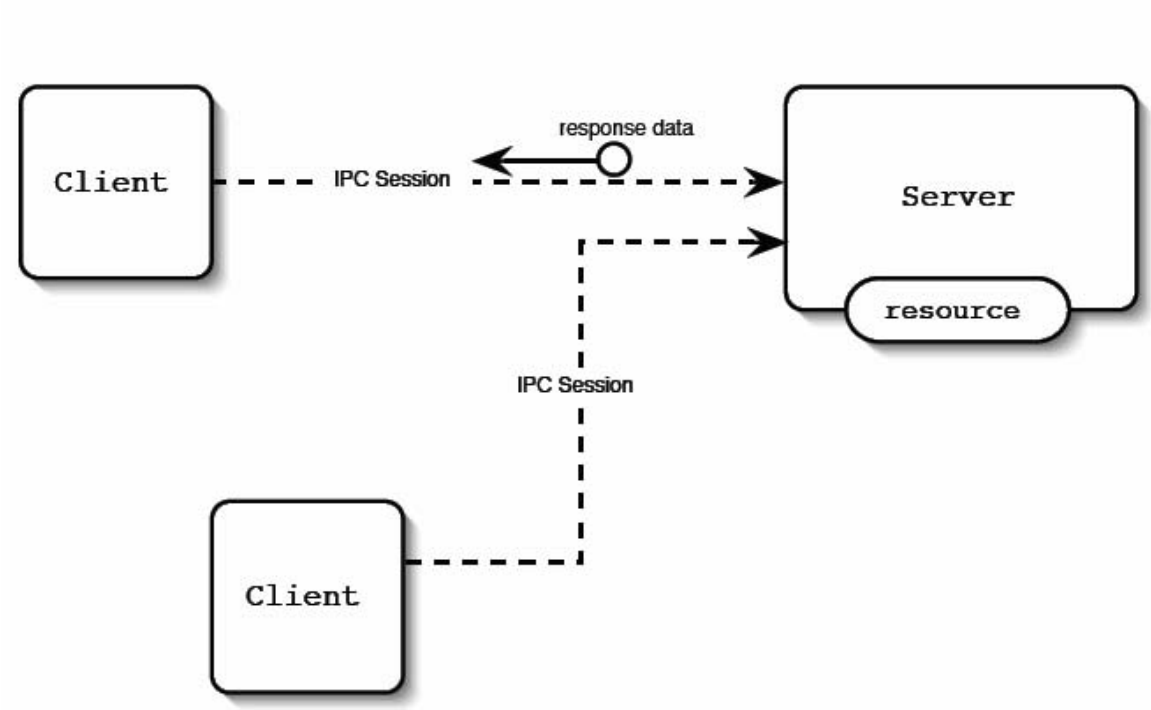
Symbian OS so far has offered semaphores, shared memory, mutexes and client/server IPC mechanisms. Since the whole OS was architected from the ground up around event-driven user-initiated interactions, the communication mechanism of choice for most user-space components was client/server session-based IPC.

The focus has always been on providing responsiveness to human users, while enabling many components and applications to share services and resources. These services and resources were always accessed through user-side servers that mediated access to them; hence the ubiquitous use of the client/server IPC.

Starting from the early 80s, Symbian OS and its 16-bit EPOC predecessor were designed *for people to use and interact with*, as opposed to most if not all other embedded and real time OSes out there, that have been used without UIs in data planes or control systems. There was typically a lot of user-initiated I/O so CPU cycles were precious and reserved for the user. In such an environment many applications were using many services simultaneously and all interactions were asynchronous and non-blocking, thus ensuring responsiveness for the user.

Current state with client/server session-based IPC

In the original and ubiquitous Symbian OS mechanism of client/server IPC, clients connect to servers by name and first establish a session that then provides the context for all further communication between clients and server. Such session-based communication comprises of client requests and server responses. Within the kernel, which mediates all messages, such request-responses are associated through session objects. So that for each session the kernel delivers messages, which the server can retrieve from its clients' space (through kernel mediation), process and then return. Finally, the server notifies the client that their request is complete. Due to its connection-oriented session-based communication, client/server is a guaranteed request completion mechanism.



For communication to take place in this way, a session has to first be established in the kernel, with a guaranteed number of message slots for the server. This connection in EKA1 is always synchronous as is the disconnection from the server. Session-based communication ensures that all clients will be notified in the case of an error or shutdown of a server, as well as that all resources will be cleared when something goes wrong or when a client disconnects or dies.

This type of communication paradigm is good for when *many clients need to reliably access a service or shared resource concurrently*. In that case the server *serialises and mediates access* to the service accordingly. It is always the case where the *clients will have to initiate a connection to the server* and the server is there to respond on demand; thus their relation is *one-to-one but not peer-to-peer*.

Limitations

Although Symbian OS client/server IPC serves its lightweight micro-kernel based architecture [1] well for client services, there are some limitations that may become increasingly taxing as the platform grows in complexity and purpose. Symbian OS, as was said, was created to serve mostly user-initiated I/O. As we have moved through to always-connected communicating devices, many more interactions and much more I/O are taking place that are not always user-initiated. For these the evolution of the architecture caters in its latest versions of EKA1 and now in EKA2.

Such limitations are manifested in certain system-level usage scenarios of course, e.g.

- Clients must know which server provides the service they want.
- IPC requires that permanent sessions between clients and server is maintained.
- Potential for deadlocks due to synchronicity of session creation and teardown if circular connections are formed.
- It is not really suitable for event multicasting.

- In general although delivery is guaranteed, there is no guarantee of real-time deterministic delivery.

Beyond Client/Server Sessions

In order to overcome such limitations and enrich the IPC mechanisms to support more paradigms of component interactions Symbian OS now (from Version 8.0) adds the following:

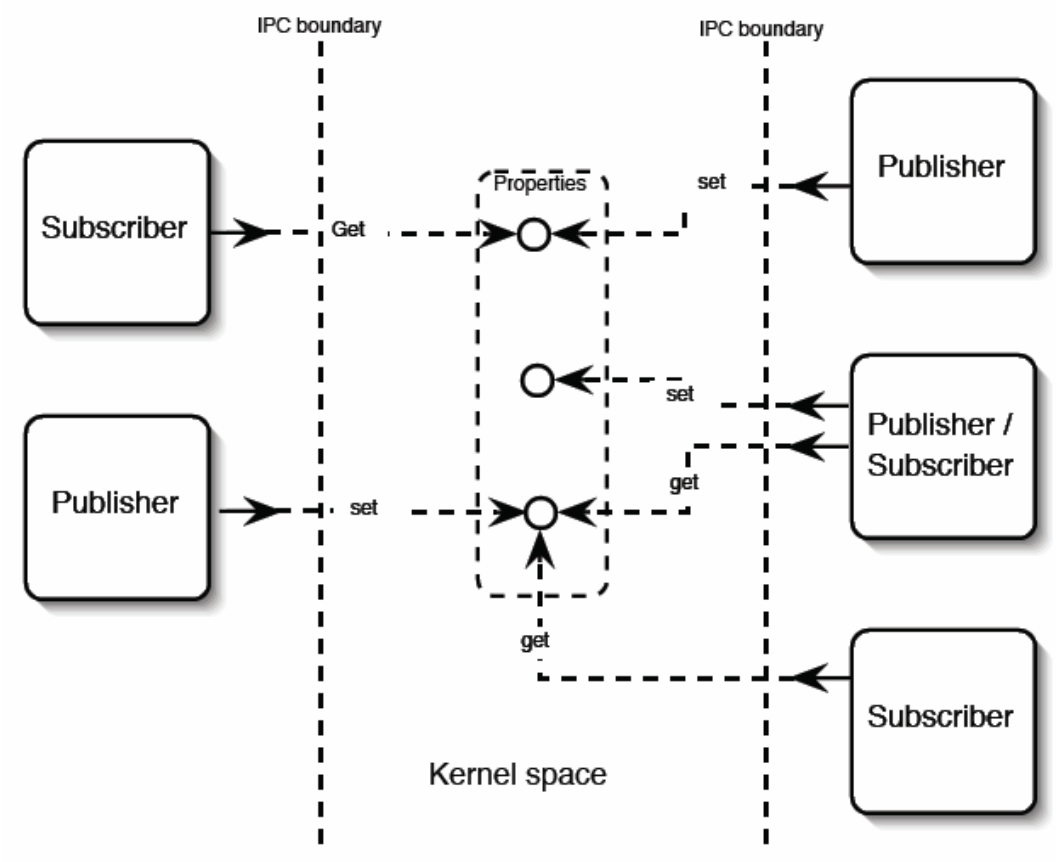
- Publish & Subscribe
- Message Queues
- Shared Buffer I/O

These new mechanisms serve very well the porting of components from other embedded and real-time OS as well as adding flexibility for refactored Symbian OS components.

Publish & Subscribe

Publish & Subscribe (P&S), is a new IPC mechanism also known as ‘properties’ that provides a means to define and publish system-wide ‘global variables’. Such properties are communicated to more than one interested peer asynchronously.

This Publish & Subscribe API can be used by both user- and kernel-side programs, via similar APIs, and thus can also provide asynchronous communication between user- and kernel-side code. In the following only the user-side usage is discussed.



Threads with P&S may have the role either of the publisher or of the subscriber, while anyone can be the one which *defines the property* to be published. Such properties are single data values, uniquely identified with an integral key. Properties have *identity* and *type*. The identity and type of a property is the only information that must be shared between a Publisher and a Subscriber – there is no need to provide interface classes or functions, though that may often be desirable in many designs.

There are six basic operations that can be performed on properties:

Define: Create a property variable and define its type and identity

Delete: Remove a property from the system

Publish: Change the value of a property

Retrieve: Get the current value of a property

Subscribe: Register for notification of changes to a property

Unsubscribe: Deregister for notifications of changes

Of these operations, the first two *need to be coupled in the same thread*, as it is only permissible for the defining thread to delete a property. Once defined, the property persists in the kernel until Symbian OS reboots or the property is deleted. Subsequently its lifetime is not tied to that of the thread or process that defined it. When properties are deleted any outstanding subscriptions for this property will be completed to the subscribers, indicating that the properties cannot be found (anymore).

As said, either publishers or subscribers may define a property, so it is not required that a property be defined before it is accessed. This paradigm allows for lazy definition of properties, hence publishing a property that is not defined is not necessarily a programming error.

For some operations such as publication and retrieval of properties, there are two modes of transaction. Although P&S operations are as far as applicability is concerned connection-less; in fact they can be used either transiently or after having set up the association between user and property, in the kernel. Thus publishers and subscribers may ‘attach’ to properties prior to operation.

Subsequently properties can be published or retrieved either by using a previously attached handle, or by specifying the property category and key with the new value. In EKA2 systems, the benefit of the former is the deterministic bounded execution time, suitable for high-priority, real-time tasks; with the exception of publishing a byte-array property that requires allocating a larger space for the new one.

In addition, properties are read and written atomically, so it is not possible for threads reading the property to get an inconsistent value or multiple for simultaneous published values to be confused.

As far as subscriptions and cancellations are concerned, clients of P&S must register their interest, by attachment, prior to usage. This is due to the asynchronous nature of the notification. For the kernel to recognize deterministically which threads need to be notified of a change in some property, that interest needs to be first associated to the property.

Notification of a property update happens in effectively four stages. Initially a client *registers* its interest in that property. Then, upon a *new publication* of that property’s

value the *client gets notified* (by completion of its subscription request). Finally after notification the *client initiates the retrieval* of the updated property.

Message Queues

Contrary to the connection-oriented nature of client/server IPC, message queues offer a *peer-to-peer many-to-many communication paradigm*. With the Message Queue API, threads may send messages to interested parties without needing to know if any thread is listening. Moreover the identity of the recipient is not needed for such communication.

There are five basic operations that can be performed on a message queue.

- Creating/opening a message queue
- Sending a message
- Receiving a message
- Waiting for space to become available in the queue
- Waiting for data to arrive in the queue

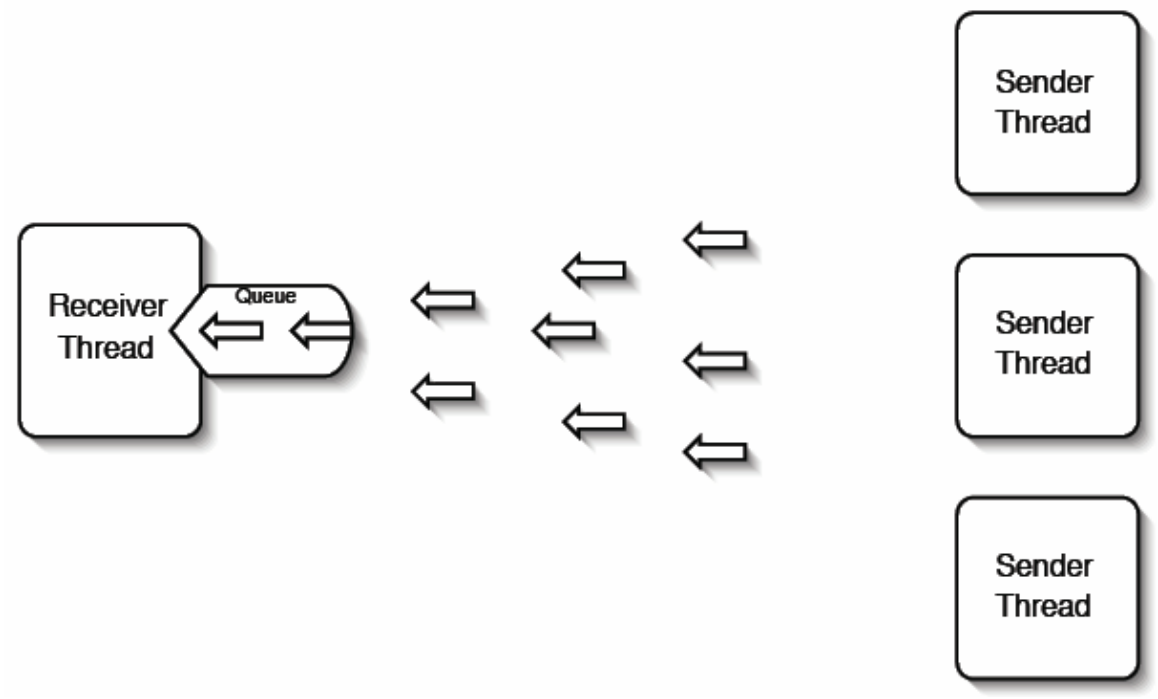
In this IPC mechanism, threads send and receive messages to and from kernel-managed objects that the queues are. In this manner messages are placed on to the queues, which are *dimensioned at the time of their creation*. Subsequently queues may run out of space, in which case senders can either block on sending or be notified that a queue overflow would otherwise occur and thus be allowed to retry later. Writers have therefore the choice, as to which message sending semantics to employ. Dimensioning at the time of creation allows for *real-time deterministic delivery* of messages in EKA2, for both sending and receiving of messages.

Message queues can either be named and globally visible to all processes, local to the current process only, or anonymous but globally accessible. Global anonymous queues (and their handle sharing between threads for that matter) are only applicable on EKA2.

Although it is also possible for queues to have multiple readers, such an arrangement in most cases needs to be combined with some sort of out-of-band collaboration between the reader threads.

Message queues effectively allow for *fire-and-forget* communication semantics between threads. Readers *are guaranteed the delivery of messages on to the queues*, but delivery to final recipients isn't. Moreover, *neither messages nor queues are persistent*; they are cleaned up when the last handle to the queue is closed. On message reception, readers can either block indefinitely until a message arrives on to the queue or be notified immediately if there is no message pending for reception, and thus retry later.

Both send and receive operations effectively *copy the message structures* to and from the kernel while local queues can be used between threads of the same process to communicate messages that point to memory mapped to their process; thus they achieve better data throughput.



A thread may also discover from a queue whether it has space available to send it messages as well as whether any messages are pending for delivery. Moreover a thread can interrogate the queue as to its message size. Additionally a thread may choose *to wait on a queue* until data or space become available.

With the Symbian OS Message Queue API, queues can also be typed so that it is possible to send and receive messages of a specific type only. This paradigm allows for type-safe IPC between participants without the overhead of type checking and the potential errors that may arise.

In general, since the size of the messages that can be exchanged in this API is relatively small (<256bytes), queues lend themselves nicely to timely event notification from many sources to one recipient as well as for passing of memory descriptors and pointers between threads in the same process. This latter usage can be employed effectively for processing intensive and/or multimedia applications.

Shared Buffer I/O

Shared buffer I/O really deserves discussion similar to that of device drivers for EKA1 [2], but will not be discussed further in this text. This IPC mechanism is really intended for device developers that need to communicate data from and to drivers.

In shared buffer I/O device drivers don't need to have a buffer of their own, but can share a buffer with a user space process. This arrangement allows the driver and its client to access the same memory without copying, while it is safe to access the buffers during interrupt handling.

Usage scenarios of the new IPC mechanisms

Below are presented some usage scenarios of how the new IPC mechanisms are likely to prove of value.

Client/Server IPC

Client/server as an IPC mechanism has been discussed many times especially in Symbian Press texts [3], [4]. Below is a graphical depiction of its usage for completeness.

In the above outline, the client-server relation between participants is clear. The client has to initiate requests that the server has to service. Moreover the server may receive such requests from many clients, in which case requests will be serialised and answered one at a time. There are no deterministic guarantees in this scenario and for every communication channel a 'connection' has to be set up. This is a typical scenario for when some kind of resource which has inherently serial access can be offered to many clients simultaneously. Thus the server's responsibilities are to *mediate access to resources* through *pre-defined service interfaces*.

Publish & Subscribe

Publish and subscribe is probably the most important new IPC mechanism and the one with the most impact. It was created to solve the problem of asynchronous multicast event notification and to allow for connection-less communication between threads.

With this new API, interested parties *do not need to know who the event provider is*. This design is essential in avoiding having many components knowing about many other components and having to connect to them, just to discover and consume events. Thus this API breaks the need to know and link to many client APIs; participants only need to know about the P&S API and the properties they are interested in.

Producers and consumers of events can now dynamically join and leave the 'conversation' without any prior commitment or connection set-up between each other. This paradigm lends itself perfectly to agent type scenarios, where both asynchronicity and autonomy are essential. Architecturally, for developers, this is very significant since their components can be *designed and deployed without any prior knowledge of who their consumers or producers may be*; nor need they supply interfaces to them other than the specification of the properties in question.

Effectively, as depicted above, publishers and subscribers are completely shielded from each other. This is the reason why multicasting is so easy and comes for free. Publication of events is referred to in this text as multicasting instead of broadcasting because it doesn't have by default to pollute the communication channels of every thread should one not wish it to. Publish & Subscribe is the best paradigm to multicast events to interested parties without needing to know explicitly who, where or how these parties will receive these events, nor necessarily involving them if they're not interested.

Accounting and resource management is handled by the kernel and thus the *complexity of usage is kept really low*. Consequently in this mode of communication, messages are not guaranteed in terms of their timely reception from remote parties, although delivery to and from the kernel is guaranteed, as well as being real-time and deterministic (in the case of EKA2). This is because remote parties may not even exist or be interested in the communication, while there is no pre-defined way to know (other than out-of band communication between the parties).

It is therefore advised that P&S should be used when a component needs to supply (or consume) timely but transient information to an *unknown number and kind* of interested parties, while maintaining maximum *decoupling* from them.

Such typical scenarios would be the dissemination of battery status information *across* the system or the publication of the status of communication links. In general whenever such contextual information is to be communicated across the system, P&S is the paradigm to use.

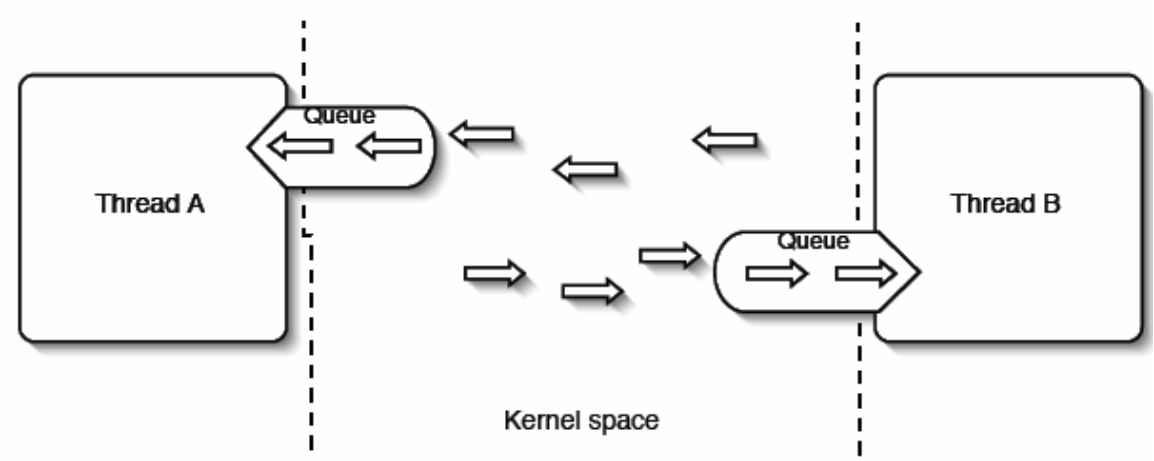
Message Queues

Message Queues is an API that allows designers to break the connection while keeping the communication between two or more threads. In that sense two threads may communicate with one another without needing to set up connections to each other. This *in effect breaks the synchronicity in the setting up and tearing down of connections* as well as the rendezvous of these threads for such set-up to take place.

Breaking such connection synchronicity has many benefits for scenarios where otherwise deadlocks would arise in using the client/server paradigm. Such deadlocks manifest themselves during cancellation or set-up of sessions between clients to servers which are themselves clients to other servers. Moreover this IPC is less alien to developers porting from other OSes to Symbian OS, where similar mechanisms (such as mailboxes) exist.

As outlined above, queues allow for many senders to communicate messages through a queue. In effect the ultimate recipient of messages is not known in a global queue, since any thread may read from it. Consequently, *delivery to the final recipient is not guaranteed*. For this to be possible, participants have to agree on an implicit protocol where a queue may be accessed from only one reader. This is Symbian's recommendation for this IPC for almost all cases. It has to be noted that *messages are sent to queues* not to final recipient threads. Therefore messages don't have any header payload stating their final destination.

Symbian OS Message Queues are not pipes. To emulate pipes in Symbian OS using the Message Queue API one has to build a protocol and abstractions on top of the API. Since messages are anonymous and don't have delivery addresses, *queues may be used as half-duplex pipes*. It is down to the communicating threads to define this protocol between them and set up the two queues necessary for peer-to-peer communication between them.



When this is done, two threads may asynchronously communicate to each other without necessarily linking to each other or client APIs other than the Message Queue one. This is beneficial since (components and) threads become *replaceable at run time* to be

either senders or receivers of messages from a queue. Thus *high availability services can be offered and be serviced at run time without disrupting their clients*.

Whereas P&S is excellent for notifications of state changes (that are inherently transient), queues are good for allowing information to be communicated and transcend the sender's state or lifetime. For example a logging subsystem could utilise a queue to receive messages from many threads that may or may not be still running at the point where the messages are read and processed. In that sense a service provider in this paradigm (as opposed to client/server) is not the one which mediates (and disseminates data) from the resource, but the one which consumes them.

How to get started

You can make use of the new public IPC mechanisms, namely P&S and Message Queues, using any C++ SDK based on Symbian OS v8.0 or higher (in particular the recently released Nokia Series 60 2nd Edition SDK Supporting Feature Pack 2). The APIs are extensively documented in the accompanying Symbian Developer Library, both in the API Reference and in the Symbian OS Guide sections. Both of the APIs are implemented in euser.dll, so no extra linking needs to be done. Note that since no SDKs (or phones) have yet been released taking advantage of EKA2, you will not yet be able to take advantage of the real-time guarantees referred to above.

In fact, although these APIs were introduced as part of Symbian OS v8.0 they were in fact back-ported to the enhanced release of Symbian OS v7.0s, on which all Nokia Series 80 and Series 90 2nd Edition SDKs were based, as well as all Series 60 2nd Edition SDKs supporting Feature pack 1 or higher. All phones based thereon also of course support the new IPC mechanisms; in practice this means *all phones based on Symbian OS v7.0s with the exception of the Nokia 6600*. So you should be able to make full use of the new IPC mechanisms working with the appropriate SDK. The one caveat is that although all necessary headers and libs are available on the SDKs, the Version 7.0s Symbian Developer Library does not include documentation for the back-ported APIs. You can however conveniently obtain the Version 8.0 Symbian Developer Library from Symbian Developer Network.

References:

[1] 'Crossing the Userland', John Pagonis, Symbian Developer Network, March 2003, http://www.symbian.com/developer/techlib/papers/userland/userland.pdf
[2] 'Overview of Symbian OS Hardware Interrupt Handling', John Pagonis, Symbian Developer Network, March 2004, http://www.symbian.com/developer/techlib/papers/HWinterupt/HwInterrupt.pdf
[3] 'Symbian OS C++ for Mobile Phones', Richard Harrison, 2003, Wiley
[4] 'Symbian OS Explained', Jo Stichbury, 2004, Wiley