

Publish and Subscribe

Mark Shackman

Version 1.0 – October 2005

1. Introduction

On any modern operating system, there is often a need for threads to communicate with each other. This is usually achieved via inter-process communications methods (IPCs). Symbian OS v9 introduces new functionality called “Publish and Subscribe” that allows the setting, retrieving and monitoring of system-wide variables and provides a new IPC mechanism for peer to peer communication between threads.

This paper describes the Publish and Subscribe functionality, including comprehensive code examples and covering both usage patterns and general usage guidelines.

2. Introducing Publish and Subscribe

The system-wide variables are usually known as ‘properties’, hence the Publish and Subscribe API is the `RProperty` class, which is defined in `e32property.h` and requires linking against `euser.lib`. The mechanism has three components:

- properties – the system-wide variables
- publishers – threads that define and set (i.e. update) a property
- subscribers – threads that retrieve the value of a property and which may also listen for changes to a property

2.1. Properties

Properties have two attributes: identity and type. These are the only pieces of information that need to be shared between a publisher and a subscriber.

2.1.1. Identity

A property is identified using a 64-bit integer, which comprises two 32-bit parts: the category and the key.

- The category is a standard UID and specifies the category to which the property belongs.
- The key uniquely identifies a property within the category, and its value depends on how keys within the category are enumerated – it can be a simple index value or another UID if the category is designed to be extensible.

2.1.2. Type

A property is a single data variable and may be either a 32-bit integer or a byte array (a descriptor) of up to 512 bytes. (The limit allows a real-time guarantee to be made.) The API also allows a byte array to be treated as being Unicode text.

Once defined, the type of a property cannot be changed.

The size of a byte array can be specified when the property is defined, so that memory is allocated initially rather than when the thread is running. The size of a byte array can be altered at any point, provided that the

maximum length (RProperty: : KMaxPropertyArray) is not exceeded. Note that if the size of a byte array is increased, memory allocation may occur for which no guarantee can be made for the time taken. Similarly, the RProperty: : ELargeByteArray property type can never provide any real-time guarantee.

3. Publish and Subscribe Operations and Code Examples

There are six operations that can be performed on a property.

3.1. Defining a property

A property is defined using the RProperty: : Define() API, which creates the variable and specifies its type and access controls. The property doesn't need to have been defined before it is accessed; this allows programming patterns where both publishers and subscribers can define the property.

Once a property has been defined, it will persist in the kernel until either it is deleted explicitly or the system reboots. Since the property's lifetime is not linked to the defining thread or process, it is necessary to check the return value from RProperty: : Define(), in case the property was defined previously but not deleted.

The following Symbian OS v8 example code shows the definition of integer and byte array properties:

```
const TUid KMyPropertyCat = {0x10012345};
enum TMyPropertyKeys {EMyPropertyInteger, EMyPropertyArray};

// define integer-type property
TInt err = RProperty: : Define(KMyPropertyCat, EMyPropertyInteger, RProperty: : Eint);
if (err != KErrAlreadyExists)
{
    User: : LeaveIfError(err);
}

// define byte array-type property of size 25
err = RProperty: : Define(KMyPropertyCat, EMyPropertyArray, RProperty: : EByteArray, 25);
if (err != KErrAlreadyExists)
{
    User: : LeaveIfError(err);
}
```

3.1.1. Security issues

Secure implementations of Symbian OS (i.e. v9 and later) require properties to be defined with security policies and to obey security rules regarding the category. See section 6 below for further details.

3.2. Deleting a property

A property may only be deleted by the defining thread (identified by the SID returned by RProcess: : Identity()), using the RProperty: : Delete() API, as follows:

```
err = RProperty: : Delete(KMyPropertyCat, EMyPropertyInteger);
if (err != KErrNotFound)
{
    User: : LeaveIfError(err);
}
```

When a property is deleted, any outstanding subscriptions will be completed with KErrNotFound.

3.3. Creating handles to properties

Properties can be accessed either by specifying the property's category and key, or by using a handle that has previously been attached to the property. Using a handle guarantees a bounded execution time, making it suitable for high priority, real time tasks (except when allocating additional memory to a byte array).

The following code shows how to create a handle to a property:

```
// create handle to integer-type property
RProperty integerProperty;
```

```
TI nt err = integerProperty.Attach(KMyPropertyCat, EMyPropertyInteger, EOwnerThread);
User::LeaveIfError(err);
```

Attaching to an undefined property is not necessarily an error. The call will succeed, but memory allocation will be required, so in this case there are no real-time guarantees.

The handle should be released using the `Close()` API when it is no longer required. Note that this doesn't cause the property to be deleted.

3.4. Publishing a property's value

A property is published (i.e. its value is updated) using the `Set()` function. This writes the new value atomically, thus ensuring that access by multiple threads is handled correctly.

When a property is published, all outstanding subscriptions are completed, even if the value is unchanged. This allows the property to be used as a simplified broadcast notification service.

Properties are published as follows:

```
// publish property using category and key
TI nt newVa lue = 0;
TI nt err = RProperty::Set(KMyPropertyCat, EMyPropertyInteger, newVa lue);
User::LeaveIfError(err); // invalid if publishing an undefined property is OK

// create handle to integer-type property
RProperty integerProperty;
TI nt err = integerProperty.Attach(KMyPropertyCat, EMyPropertyInteger, EOwnerThread);
User::LeaveIfError(err);

// publish property using handle
TI nt err = integerProperty.Set(++newVa lue);
User::LeaveIfError(err); // invalid if publishing an undefined property is OK

integerProperty.Close();
```

Publishing an undefined property is not necessarily an error; in this case, the value returned from `Set()` should be checked and processed.

Note that, before publishing a property, the security policy defined when the property was created is checked, and the appropriate action taken if the check fails.

3.5. Retrieving a property's value

A property is retrieved using the `Get()` function. This reads the new value atomically, thus ensuring that access by multiple threads is handled correctly.

Properties are retrieved as follows:

```
// retrieve a property using category and key
TI nt newVa lue;
TI nt err = RProperty::Get(KMyPropertyCat, EMyPropertyInteger, newVa lue);
User::LeaveIfError(err);

// create handle to integer-type property
RProperty integerProperty;
TI nt err = integerProperty.Attach(KMyPropertyCat, EMyPropertyInteger, EOwnerThread);
User::LeaveIfError(err);

// retrieve a property using handle
TI nt err = integerProperty.Get(newVa lue);
User::LeaveIfError(err);

integerProperty.Close();
```

As for publishing, retrieving an undefined property is not necessarily an error; in this case, the value returned from `Get()` should be checked and processed. Again, the security policy is also checked before the property is retrieved.

3.6. *Subscribing to a property*

Subscribing to a property allows a thread to make an asynchronous request to be informed of any updates to a property. Calling `RProperty::Subscribe()` on a previously attached property object will inform the caller when the property is next updated. The caller will need to resubscribe to get subsequent notifications, and to retrieve the property's value if required. This is shown in the following code example.

```
const TUid KMyPropertyCat={0x10012345};
enum TMyPropertyKeys {EMyPropertyInteger, EMyPropertyArray};

// attach to the property
RProperty integerProperty;
TInt err= integerProperty.Attach(KMyPropertyCat, EMyPropertyInteger, EOwnerThread);
User::LeaveIfError(err);

// wait for the attached property to be updated
TRequestStatus status;
integerProperty.Subscribe(status);
User::WaitForRequest(status);

// Notification complete, retrieve the value.
TInt count;
integerProperty.Get(count);
. . .
```

It is possible that the property may be updated before the subscriber has had an opportunity to process the property value, so the subscriber may become out of synchronisation with the property. In this case, before processing a subscription completion, the subscriber should use an active object to reissue the subscription request immediately. This is shown in the following code example:

```
const TUid KMyPropertyCat={0x10012345};
enum TMyPropertyKeys {EMyPropertyInteger, EMyPropertyArray};

// Active object that tracks changes to the integer property
class CPropertyWatch : public CActive
{
    enum {EPriority=0};
public:
    static CPropertyWatch* NewL();
private:
    CPropertyWatch();
    void ConstructL();
    ~CPropertyWatch();
    void RunL();
    void DoCancel();
private:
    RProperty iProperty;
};

CPropertyWatch* CPropertyWatch::NewL()
{
    CPropertyWatch* self = new (ELeave) CPropertyWatch;
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop(self);
    return self;
}

CPropertyWatch::CPropertyWatch()
: CActive(EPriority)
{
}

void CPropertyWatch::ConstructL()
{
    User::LeaveIfError(iProperty.Attach(KMyPropertyCat, KMyPropertyInteger));
    CActiveScheduler::Add(this);
    // initial subscription and process current property value
    RunL();
}

CPropertyWatch::~CPropertyWatch()
{
    Cancel();
    iProperty.Close();
}
```

```

void CPropertyWatch::DoCancel ()
{
    iProperty.Cancel ();
}

void CPropertyWatch::RunL()
{
    // resubscribe before processing new value to prevent missing updates
    iProperty.Subscribe(iStatus);
    SetActive();

    // property updated, get new value
    TInt value;
    if (iProperty.Get(value) == KErrNotFound)
    {
        // property deleted, do necessary actions here...
        PropertyDeleted();
    }
    else
    {
        // use new value ...
        PropertyChanged(value);
    }
}

```

Note that a subscription request on an undefined property will not complete until the property is defined and published. If the subscriber doesn't have sufficient capability, the request will complete (either immediately or when the property is defined) with `KErrPermissionDenied`.

3.7. *Unsubscribing from a property*

A thread can unsubscribe from a property (i.e. cancel an outstanding request) by calling the `RProperty::Cancel()` function.

4. Publish and Subscribe Patterns of Usage

There are three common patterns of usage for Publish and Subscribe, as outlined below.

4.1. *Standard State*

The standard state pattern is used for important events that are used widely within the system, such as signal strength.

The publisher defines the property, ensuring that sufficient memory is reserved for all possible sizes of arrays and then ignoring a `KErrAlreadyExists` error code. Property values are then published as required, with failed calls to `RProperty::Set()` being treated as serious errors, as they indicate a failure to publish the data. Appropriate responses might be to panic or reboot the system.

Subscribers use the API as normal to retrieve updates to the property.

Since sufficient memory to hold all possible sizes of an array property was allocated when the property was defined, the property can continue to be published, even if the system enters an out-of-memory situation. This memory is permanently allocated even if there are no subscribers to the property.

4.2. *Pure Event Distribution*

The pure event pattern is used when events, rather than values, need to be distributed.

The publisher should use an integer property, to ensure that the minimum amount of memory is wasted. The value that is published should not have significance; even if the value is not changed, subscribers will still be notified that a value has been published and can take appropriate action.

4.3. *Speculative Publishing*

The speculative publishing pattern is used when it isn't known how widely used a property will be (if at all).

Using the standard state pattern is wasteful of memory if the property is unused by subscribers. In the speculative pattern, the publisher doesn't call `RProperty::Define()`, but just calls `RProperty::Set()` as required and ignores any `KErrNotFound` errors.

Instead, it is the subscriber which calls `RProperty::Define()`, creating the property and allocating the memory needed to store the value. Errors of `KErrAlreadyExists` should be ignored, as this just shows that another interested subscriber has already defined the property. The first call to `RProperty::Get()` that the subscriber makes may return the default value of the property (zero or a zero length descriptor), so this should be handled sensibly.

Although the speculative publishing pattern has the advantages of not allocating memory unless the property has a subscriber and making the publisher code simpler, some time may still be wasted on publishing values needlessly. This should be of concern only where the value is published very frequently. At the opposite end of the scale, if the value is published very infrequently, the subscriber could have a dummy value for a long time and this pattern is inappropriate if the dummy value is not meaningful.

5. Publish and Subscribe Usage Guidelines

These guidelines outline usage patterns that can improve performance when using the Publish and Subscribe user-side API.

5.1. *Use a handle*

For more frequent calls to the `Get()` and `Set()` functions, attach to the property (using the `Attach()` function) beforehand. The handle-based variants of `Get()` and `Set()` are constant time operations, executing much faster than those variants taking UID and key parameters.

5.2. *Preallocate memory only if publishing is time-critical*

Preallocating memory for array-based properties avoids memory allocations, but wastes space if the data is shorter than the reserved space. `Set()` automatically extends the array if required, so the only reason to preallocate the memory is if the `Set()` operation has to have a known execution time (i.e. be real time).

5.3. *Consider using Speculative Publishing*

Even when standard state seems to be appropriate, it is worth considering using speculative publishing, particularly for low-level components that are unaware of how they are used or how the wider system configuration may affect them.

5.4. *Consider the update frequency*

Whenever a property changes, subscribers are notified and their threads run to service the notification. If the property changes often, knowing the frequency of the changes may allow subscribers to avoid performing substantial, wasteful processing each time.

For example, a property that publishes signal strength could be updated many times each second. If a large number of subscribers used this to do substantial processing, device responsiveness and battery life could be severely affected.

A possible solution would be to consider the number of possible subscribers and what information they actually require. In the above example, it may be better to define another property or set of properties that publish data less frequently (such as every 2 seconds) or only on certain conditions (such as losing/gaining a signal). Subscribers could then select the most appropriate property to subscribe to.

6. Publish and Subscribe Security Issues

6.1. Defining properties

We noted above that properties are defined in terms of a category, which is a UID, and a key. Under Symbian OS v9.1, to ensure that processes are “caged” to avoid one process defining another process’ property, the category UID must match the SID of the process that is defining the property. The recommended variant of the `Define()` function shown below therefore does not have a parameter for the category – instead, it takes the category to be equal to the SID of the process.

```
static IMPORT_C TInt Define(TUint aKey, TInt aAttr, const TSecurityPolicy &aReadPolicy, const
TSecurityPolicy &aWritePolicy, TInt aPreallocated=0);
```

For backwards compatibility, there is a variant of the `Define()` function that does take a parameter for the category and is available in Symbian OS v9 and later. To use this, the caller must have the `WriteDeviceData` capability and the category value must be less than `KUIDSecurityThresholdCategoryValue`, as defined in `e32property.h`. Use of this variant is not recommended.

6.2. Reading and writing properties

Most of the Publish and Subscribe APIs require a category parameter. This should be identical to the SID value defined in the MMP file. This value is compared with the SID value obtained automatically when the property was defined, providing a security check.

Secure implementations of Symbian OS also require properties to be defined with two security policies – one to specify processes that can publish the property value and the other to specify the subscribers. The security policies are `TSecurityPolicy` objects, although in practice the `_LIT_SECURITY_POLICY_...` macros are used for efficiency. These generate constant objects that behave like `TSecurityPolicy` objects and the `TSecurityPolicy` documentation in the Developer Library provides further details.

The code below illustrates defining a property with security policies. Subscribers are unrestricted, but publishers must have the power management system capability `TCapability: ECapabilityPowerManagement`.

```
const TUid KMyPropertyCat = {0x10013579}; // same as UID3/SID in MMP
enum TMyPropertyKeys {EMyPropertyInteger, EMyPropertyArray};

static _LIT_SECURITY_POLICY_PASS(KAllowAllPolicy);
static _LIT_SECURITY_POLICY_C1(KPowerMgmtPolicy, ECapabilityPowerMgmt);

// define integer-type property, restricted publisher, unrestricted subscriber
TInt err = RProperty::Define(EMyPropertyInteger, RProperty::EInt, KAllowAllPolicy,
KPowerMgmtPolicy);
if (err != KErrAlreadyExists)
{
    User::LeaveIfError(err);
}
```

Note that any outstanding subscriptions on a property may, at the point of definition, be completed with `KErrPermissionDenied` if they fail the security policy check.

7. Migrating From System Agent

The System Agent component of Symbian OS published a set of system variables, which client processes could connect to to receive notification of changes to the variables or to set new values. This API was removed in Symbian OS v9 and the system variables (i.e. properties) are now available through the Publish and Subscribe mechanism. Note that there is no equivalent available for the System Agent conditional notification mechanism.

Users of the System Agent therefore need to migrate their applications according to the following guidelines:

- All system properties reside in the System Category, `KUIDSystemCategory`. The properties are enumerated in the file `sacls.h` and are created with pre-defined capability-based security policies which

protect the Set () function. The header file also contains details of the capabilities required by these security policies.

- The following header files are no longer exported, so references to them must be removed:
 - saclient.h**
 - sairas.h**
 - savarset.h**
- The library **sysagt.dll** is no longer built, so references to **sysagt.lib** must be removed from MMP files
- References to **sysagent.iby** must be removed from IBY and OBY files
- The System Agent Messaging Observer is no longer supplied, so references to **sysamob.exe** and **sysamob.iby** must be removed

8. Conclusion

Publish and Subscribe is a welcome and integral addition to Symbian OS, both in terms of access to global variables and in extending the range of IPC mechanisms. Although inaugurated in Symbian OS v9, Publish and Subscribe has since been back-ported to Symbian OS v8.x, due to its popularity.

Glossary

The following technical terms and abbreviations are used within this document.

Term	Definition
Capability	Defines access to restricted APIs
MMP	Application project file
SID	Secure Identifier
UID	Unique Identifier

[Back to Developer Area](#)

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.