|  |  |
| --- | --- |
| Contributor team | Contributor(s) |
| Kernel & Hardware Services/ Platform Independent |  |

Change history

| Version | Date | Status | Comments |
| --- | --- | --- | --- |
| 1.0 | 30.09.2010 | Published |  |

Table of contents

[1. Document control 4](#_Toc273964123)

[1.1 References 4](#_Toc273964124)

[1.2 Abbreviations and definitions 4](#_Toc273964125)

[2. Introduction 5](#_Toc273964126)

[2.1 Purpose and Scope 5](#_Toc273964127)

[2.2 Domain Manager Notification Interface Overview 5](#_Toc273964128)

[3. Domain Manager Notification API 6](#_Toc273964129)

[3.1 Questions & Answers 6](#_Toc273964130)

[3.1.1 What are deferrals? 6](#_Toc273964131)

[3.1.2 Why would a client want to defer a state transition? 6](#_Toc273964132)

[3.1.3 What is the maximum number of deferrals available to a client? 6](#_Toc273964133)

[3.1.4 What type of transitions can use this transition timeout behaviour? 6](#_Toc273964134)

[3.1.5 What happens if a client fails to complete a task after deferring? 6](#_Toc273964135)

[3.1.6 Which header files do I need to include? 6](#_Toc273964136)

[3.1.7 Which library do I need to link my binary against? 7](#_Toc273964137)

[3.2 Using the Domain Manager Notification interface with deferrals 8](#_Toc273964138)

[3.2.1 How do clients use the Domain Manger Notification interface with deferrals? 8](#_Toc273964139)

[3.2.2 Implementing a CDomainMemberKeepAlive object 11](#_Toc273964140)

[3.2.3 CDomainMemberKeepAlive class example 11](#_Toc273964141)

[3.2.3.1 Construction 12](#_Toc273964142)

[3.2.3.2 Handling transition notifications 13](#_Toc273964143)

[3.2.3.3 Acknowledging completion of a transition 13](#_Toc273964144)

[4. Appendix 15](#_Toc273964145)

[4.1 Source code for CDomainMemberTask implementation 15](#_Toc273964146)

[4.2 Source code for CDomainMemberKeepAlive implementation 16](#_Toc273964147)

# Document control

## References

|  |  |
| --- | --- |
| Base\_How\_To\_Configure\_Domain\_Manager\_Policy\_DLLs  (os/kernelhwsrv/kernelhwsrv\_info/doc\_pub ) | Details use of Domain Manager Policy DLLs. |
|  |  |

## Abbreviations and definitions

|  |  |
| --- | --- |
| Definition | Description |
|  |  |
|  |  |
|  |  |

# Introduction

## Purpose and Scope

This document contains questions and answers and information on how to use the Domain Manager Notification interface with deferrals.

It provides example code and diagrams on how a client should use the Domain Manager Notification interface.

This document is meant as a reference for using the Domain Manager Notification interface with deferrals. This interface is required when a Domain Member might need more time to complete its work before acknowledging a state transition.

The reader is assumed to have prior knowledge of the Domain Manager component.

## Domain Manager Notification Interface Overview

The Domain Manager has been extended to support limited deferral of acknowledgements. This feature termed Shutdown Monitoring enables the Domain Manager to monitor the progress of system shutdown and to force device shutdown if any client hangs during the shutdown process.

This work involved extending the Domain Manager Notification API to support deferring domain member transition timeouts and handling of notifications.

The diagram below shows an overview of the Domain Manager, its interfaces and the typical clients present in the device. The existing platform interfaces that have been extended to support the Shutdown monitoring feature are shown in cyan while the new platform interface added to the design is shown in blue.



Figure Domain Manager Overview

# Domain Manager Notification API

## Questions & Answers

### What are deferrals?

Deferrals occur when a client i.e. a Domain Member makes a request for more time to complete its work for the state transition. The Domain Member, instead of acknowledging on receiving a state transition notification makes a request for more time.

It should be noted that deferrals are not always possible and whether the member will actually be given more time depends on if:-

* The current transition allows deferrals at all.
* The member still has deferrals left.
* The deferral request was received in time.

There is also a limit on how many times a client can defer the acknowledgement of a transition. The maximum number of deferrals is determined by the target state of the transition.

The Domain Member must also be a trusted client in order for it to make a deferral request. It must have either or both of the following capabilities:-

* WriteDeviceData.
* ProtServ.

### Why would a client want to defer a state transition?

A client might need more time to complete its work before acknowledging a state transition. For instance, a client after having received a power down transition notification might defer state transition by requesting more time. This should give the client time to complete its tasks and shutdown gracefully.

### What is the maximum number of deferrals available to a client?

This is configurable and can be set in the Domain Policy DLL. For a description of this DLL and how to configure the settings for the deferral see [1].

### What type of transitions can use this transition timeout behaviour?

The Domain Policy DLL declares and configures possible transition states. It also defines a timeout and deferral limit for each domain state. An example of this is the SSM (System State Manager) V2 domainpolicy2 component which can be seen in Figure 1 above. This policy DLL enables timeout behaviour for a Shutdown state.

### What happens if a client fails to complete a task after deferring?

The Domain Manger will consider the client as being unresponsive or unable to transition within the deferral time limits. It will then move on to the next Domain in the Domain tree to continue transitioning all the Domains in the tree.

### Which header files do I need to include?

In order to use the Domain Manager Notification API, the following header has to be included.

#include <domainmember.h>

### [Which library do I need to link my binary against?](#_Toc246916003)

The binary will need to be linked against the domaincli.lib import library. The following line should be added to the relevant client MMP file.

LIBRARY domaincli.lib

## Using the Domain Manager Notification interface with deferrals

### How do clients use [the Domain Manger Notification interface with deferrals?](#_Toc246916007)

In order to minimize development complexity for client writers wishing to utilise the new functionality, the Domain Member Notification API provides a class CDmDomainKeepAlive.

This class has been provided to give clients a simpler alternative to interfacing directly with the RDmDomain class (which contains the method RDmDomain::DeferAcknowledgement() for deferring an acknowledgment to a domain transition).

The diagram in Figure 2 below is a top level overview of how a CDmDomainKeepAlive Domain member defers a transition notification. The use of the class simplifies the implementation of the functionality required for performing deferrals as seen in stage 3 and 4 in Figure 2 below.

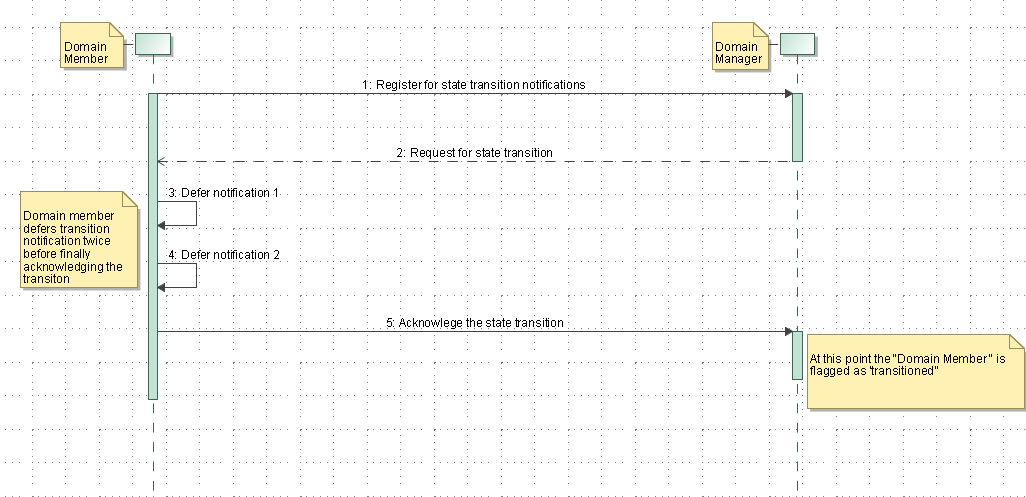


Figure 2 CDmDomainKeepAlive Domain Member deferring a transition

The diagram in Figure 3 below shows a derived class CDomainMemberKeepAlive using the interface for deferrals.

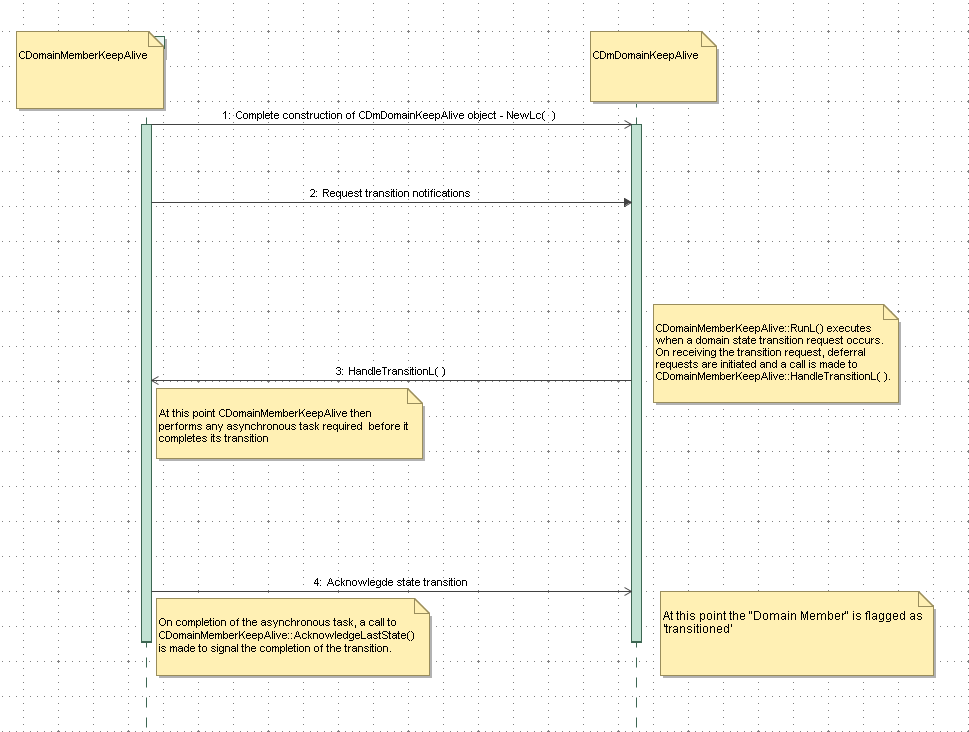


Figure 3 CDomainMemberKeepAlive object in use

An instance of the class derived from CDmDomainKeepAlive automatically request transition deferrals for as long as the client application requires and the server permits (as determined by the deferral budget for the target state, specified by the relevant Domain policy).

The class CDmDomainKeepAlive and all its interfaces are shown below:

class CDmDomainKeepAlive : public CDmDomain

{

public:

IMPORT\_C CDmDomainKeepAlive(TDmHierarchyId aHierarchyId, TDmDomainId aDomainId);

IMPORT\_C ~CDmDomainKeepAlive ();

IMPORT\_C void AcknowledgeLastState (TInt aError);

Virtual void HandleTransitionL () =0;

IMPORT\_C virtual TInt HandleDeferralError (TInt aError);

protected:

IMPORT\_C void ConstructL ();

IMPORT\_C void RunL ();

private:

CDmKeepAlive\* iKeepAlive;

TUint32 iReservedSpace [2];

};

First, note the public members: a constructor takes a Domain hierarchy (aHierarchyId) and an ID parameter (aDomainId).

IMPORT\_C CDmDomainKeepAlive (TDmHierarchyId aHierarchyId, TDmDomainId aDomainId);

These parameters are used for initialising members of the inherited class CDmDomain. The Domain Hierarchy parameter is used to identify the domain hierarchy to connect to while the Domain ID parameter is used to identify the domain to connect to.

The public destructor is used for cleaning up the internal CDmKeepAlive active object and the derived CDmDomainKeepAlive class. The CDmKeepAlive active object, once activated, repeatedly attempts to defer a transition deadline from the Domain Manager.

IMPORT\_C ~CDmDomainKeepAlive ();

Derived classes using this destructor should not need to call Cancel() as the parent class destructor ~CDmDomain() will cancel transition notifications while ~CDmKeepAlive() will cancel any outstanding deferral.

The final three public methods are the crucial ones.

HandleTransitionL() is a pure virtual function that must be implemented to provide the functionality necessary to make use of this class for deferrals.

virtual void HandleTransitionL () =0;

This function is called when the function RunL() in CDmDomainKeepAlive executes i.e. when a domain state transition request occurs. On receiving the transition request, the active object (CDmDomainKeepAlive object) starts deferring the transition. Once the deferral timeouts, another deferral request is made until the deferral limit for the domain is reached.

Clients implementing this function should add code to handle a domain transition request i.e. perform any asynchronous tasks required by the domain member before it completes its transition. The implementation of this function should be used first to call CDmDomain::RequestTransitionNotification() again. This eliminates the possibility of a transition occurring between acknowledging and registering for notification e.g. a client ready to acknowledge the last transition, and willing to register for notification of the next one, should call CDmDomain::RequestTransitionNotification() and then call AcknowledgeLastState(), immediately afterwards.

Once all asynchronous tasks are complete a call is made to AcknowledgeLastState() to signal the completion of a transition. This call should be made by the code which has just completed the asynchronous tasks.

The implementation of this function should be kept as quick as possible, any slow operations (e.g. File Server calls) should be initiated asynchronously and handled using other active objects.

AcknowledgeLastState() is a method which must be called by the client to acknowledge it has completed the state transition.

IMPORT\_C void AcknowledgeLastState (TInt aError);

Once this is done the active object will no longer attempt to defer the transition. HandleTransitionL() should not call AcknowledgeLastState() unless it can trivially determine that no action at all is required for the given transition.

aError is the error returned to the Domain Manager. The client should set this to KErrNone if it successfully transitioned to the new state or to one of the system-wide error codes.

HandleDeferralError(TInt aError) is a method called when the internal CDmKeepAlive active object leaves. The method RunError() of the CDmKeepAlive class calls this method.

IMPORT\_C virtual TInt HandleDeferralError (TInt aError);

The default implementation of this function will simply ignore errors that it presumes the client cannot or need not handle. The following are the error codes ignored:-

KErrCompletion Client has now acknowledged notification - not an error.

KErrCancel Server cancelled request - client can do nothing.

KErrNotSupported Deferral not possible - client can do nothing.

KErrNotReady Deferral too late or too early - client can do nothing.

If the client does want to handle or inspect these errors e.g. for diagnostic purposes, they can override this method. All other error codes will be returned unhandled to the active scheduler, leading to a panic.

Finally, there are two protected members.

ConstructL() is the second–phase constructor which completes the construction of a CDmDomainKeepAlive object.

IMPORT\_C void ConstructL ();

RunL() is the asynchronous event handler function for a CDmDomainKeepAlive object. This function is called by the Active Scheduler when any outstanding request to the Domain Manger server is completed.

IMPORT\_C void RunL ();

For this object, the function handles completion of request notifications and commencement of deferrals. Clients of this interface should not need to override this, they will be notified of events via calls to HandleTransitionL().

### Implementing a CDomainMemberKeepAlive object

Implementing a CDomainMemberKeepAlive object for making deferral request is quite simple. It is summarized in the following steps.

1. Create a class derived from CDmDomainKeepAlive.
2. Implement NewL() and NewLC() as normal, call ConstructL() and connect to a Domain.
3. Implement HandleTransitionL() to handle transition notifications.
4. Implement a function which makes a call to AcknowledgeLastState() once the client has completed all its asynchronous tasks i.e. tasks needing completion before transition.
5. Optionally override HandleDeferralError() to handle deferral errors.
6. Implement the destructor to handle cleanup of derived class.

### CDomainMemberKeepAlive class example

This section describes how to implement and use a class derived from CDmDomainKeepAlive. It covers the implementation of the constructor and implementation of the method HandleTransitionL().

An example class CDomainMemberTask used for running asynchronous tasks and also signaling the completion of a domain transition is also shown. This class is required to illustrate the practical use of deferrals.

The example class CDomainMemberTask used for running asynchronous tasks is shown below.

//////////////////////////////////////////////////////////////////////////////////////////////////////

//

// CDomainMemberTask

//

// This class simulates the execution of a task carried out by the domain member while

// it defers a domain transition request. The class is derived from CTimer. It uses the

// method CTimer::After (aInterval) to simulate the execution of a task by submitting a timer request

// which once completed could be repeated based on the value of iRepeatCount.

//////////////////////////////////////////////////////////////////////////////////////////////////////

class CDomainMemberTask: public CTimer

{

public:

static CDomainMemberTask\* NewLC ();

~CDomainMemberTask ();

void BeginTask (TInt aRepeat, TInt aInterval, CDomainMemberKeepAlive\* aDomainMemberKeepAlive);

void CompleteTask ();

void RunL ();

private:

CDomainMemberTask ();

private:

TInt iRepeatCount;

TInt iInterval;

CDomainMemberKeepAlive\* iDomainMemberKeepAlive;

};

An example of a class derived from CDmDomainKeepAlive is shown below.

//Forward declaration of CDomainMemberTask

class CDomainMemberTask;

//////////////////////////////////////////////////////////////////////////////////////////

//

// CDomainMemberKeepAlive

//

// This class is derived from CDmDomainKeepAlive. CDomainMemberKeepAlive

// represents a Domain Member and uses CDmDomainKeepAlive interface for

// deferring transition requests.

/////////////////////////////////////////////////////////////////////////////////////////

class CDomainMemberKeepAlive: public CDmDomainKeepAlive

{

public:

static CDomainMemberKeepAlive\* NewL (TDmHierarchyId aHierarchy, TDmDomainId aId);

~CDomainMemberKeepAlive ();

void HandleTransitionL ();

void DoTaskL (); //A method used to simulate the execution of a task

private:

CDomainMemberKeepAlive (TDmHierarchyId aHierarchy, TDmDomainId aId);

private:

TDmHierarchyId iHierarchy;

TDmDomainId iId;

CDomainMemberTask\* iMemberTask; // Pointer to the class which handles execution of a task

};

#### Construction

The code below shows the implementation of the method NewL() in CDomainMemberKeepAlive class. Here, the ConstructL() method is invoked and a request for transition notifications is made.

// Standard 2-Phase Construction

CDomainMemberKeepAlive\* CDomainMemberKeepAlive::NewL (TDmHierarchyId aHierarchy, TDmDomainId aId)

{

CDomainMemberKeepAlive\* self=new (ELeave) CDomainMemberKeepAlive (aHierarchy, aId);

CleanupStack::PushL (self);

self->ConstructL ();

self->RequestTransitionNotification ();

CleanupStack::Pop ();

return self;

}

#### 

#### Handling transition notifications

In the example below, an implementation of HandleTransitionL() is shown. On entry into the function a call is made to CDmDomain::RequestTransitionNotification().This is performed as the member is interested in the next domain transition request.

The client application then calls the method CDmDomain::GetState(), this is normally called after a notification request has completed. It then performs any application-dependent action demanded by the state, and then acknowledges the state transition.

In this case, the client makes a call to CDomainMemberTask::DoTaskL() after confirming that critical shutdown state actions are required. This method then simulates the running of an asynchronous task which needs to be performed before completing the domain transition request.

The task performed by CDomainMemberTask::BeginTask() is actually implemented by a timer event which is triggered after 1000000us.This event is repeated 4 times after which an acknowledgement of the state transition is then made by a call to the method CDomainMemberTask::CompleteTask(). For the case where the state is not EShutdownCritical, a call to AcknowledgeLastState() is made to signal the completion of the transition.

For details on the other bits of code implemented for the CDomainMemberTask class see the Appendix at the end of this document.

void CDomainMemberKeepAlive::HandleTransitionL()

{

RequestTransitionNotification();

TDmDomainState domainState = GetState();

if (domainState == EShutdownCritical) // Check if critical shutdown state actions are required

{

TRAPD(leavecode,DoTaskL());

if (leavecode)

AcknowledgeLastState(leavecode); // Acknowledge and complete transition with error code if DoTaskL() leaves

}

else

{

TInt error = KErrNone;

AcknowledgeLastState(error);

}

}

// Method to run Domain Member task

void CDomainMemberKeepAlive::DoTaskL()

{

iMemberTask = CDomainMemberTask::NewL();

iMemberTask->BeginTask(4,1000000,this); // Repeat 4 times , 1000000 us interval

}

#### Acknowledging completion of a transition

Once all asynchronous tasks are complete a call to AcknowledgeLastState() must be made. This will signal the completion of the transition by the client.

For this example, CDomainMemberTask::CompleteTask() is called in the RunL() method of the CDomainMemberTask class. This method is called when the timer event expires and the timer event repeat count value is exceeded.

//This method completes the simulation of a task. It also calls AcknowledgeLastState () to signal the completion

//of a Domain transition.

void CDomainMemberTask::CompleteTask ()

{

//Acknowledge to notify completion of transition

TInt error = KErrNone;

iDomainMemberKeepAlive->AcknowledgeLastState (error);

}

//Once all simulated tasks have completed, a call is made to CompleteTask ()

//to notify completion of a domain transition

void CDomainMemberTask::RunL ()

{

if (iRepeatCount--)

{

//-----------

// dummy task

//----------

After (iInterval); //re-issue request

}

else

{

CompleteTask (); //repeat count expired, end event loop to exit

}

}

# Appendix

## Source code for CDomainMemberTask implementation

// Standard 2-Phase Construction

CDomainMemberTask\* CDomainMemberTask::NewL()

{

CDomainMemberTask\* self=new (ELeave) CDomainMemberTask ();

CleanupStack::PushL (self);

self->ConstructL ();

CleanupStack::Pop ();

CActiveScheduler::Add (self);

return self;

}

// CDomainMemberTask Constructor. It is called from the NewL () function.

CDomainMemberTask::CDomainMemberTask (): CTimer (CActive::EPriorityStandard)

{

iDomainMemberKeepAlive=NULL;

}

//This method commences the simulation of a task. See description of CDomainMemberTask class

void CDomainMemberTask::BeginTask (TInt aRepeat, TInt aInterval, CDomainMemberKeepAlive\* aDomainMemberKeepAlive)

{

iDomainMemberKeepAlive=aDomainMemberKeepAlive;

iRepeatCount=aRepeat;

iInterval=aInterval;

CTimer::After (iInterval);

}

//This method completes the simulation of a task. It also calls AcknowledgeLastState () to signal the completion

//of a Domain transition.

void CDomainMemberTask::CompleteTask ()

{

//Acknowledge to notify completion of transition

TInt error = KErrNone;

iDomainMemberKeepAlive->AcknowledgeLastState (error);

}

//Once all simulated tasks have completed, a call is made to CompleteTask ()

//to notify completion of a domain transition

void CDomainMemberTask::RunL ()

{

if (iRepeatCount--)

{

//-----------

// dummy task

//----------

After (iInterval); //re-issue request

}

else

{

CompleteTask (); //repeat count expired, end event loop to exit

}

}

//Destructor to handle cleanup

CDomainMemberTask:: ~CDomainMemberTask ()

{

iDomainMemberKeepAlive=NULL;

}

## Source code for CDomainMemberKeepAlive implementation

// Standard 2-Phase Construction

CDomainMemberKeepAlive\* CDomainMemberKeepAlive::NewL (TDmHierarchyId aHierarchy, TDmDomainId aId)

{

CDomainMemberKeepAlive\* self=new (ELeave) CDomainMemberKeepAlive (aHierarchy, aId);

CleanupStack::PushL (self);

self->ConstructL ();

self->RequestTransitionNotification ();

CleanupStack::Pop ();

return self;

}

// CDomainMemberKeepAlive Constructor. It is called from the NewL () function.

CDomainMemberKeepAlive::CDomainMemberKeepAlive (TDmHierarchyId aHierarchy, TDmDomainId aId):

CDmDomainKeepAlive (aHierarchy, aId), iHierarchy (aHierarchy), iId (aId)

{

iMemberTask=NULL;

}

void CDomainMemberKeepAlive::HandleTransitionL()

{

RequestTransitionNotification();

TDmDomainState domainState = GetState();

if (domainState == EShutdownCritical) // Check if critical shutdown state actions are required

{

TRAPD(leavecode,DoTaskL());

if (leavecode)

AcknowledgeLastState(leavecode); // Acknowledge and complete transition with error code if DoTaskL() leaves

}

else

{

TInt error = KErrNone;

AcknowledgeLastState(error);

}

}

// Method to run Domain Member task

void CDomainMemberKeepAlive::DoTaskL()

{

iMemberTask = CDomainMemberTask::NewL();

iMemberTask->BeginTask(4,1000000,this); // Repeat 4 times , 1000000 us interval

}

//Destructor to handle cleanup of derived class.

CDomainMemberKeepAlive::~CDomainMemberKeepAlive()

{

delete iMemberTask;

}