Domino eXplorer (DX) Kernel API Reference

For DX Version: 3.14.x

Author: Ian Tree
Owner: HMNL b.*v.*
Customer: Public
Status: Final
Date: 10/03/2015 15:34
Version: 3.14.0

Disposition: Open Source

Document History

Revision History

Date of this revision: 10/03/2015 15:34	Date of next revision *None*

Revision Number	Revision Date	Summary of Changes	Changes marked
0.1	02/06/11	Initial Base Version	No
3.12.0	06/05/12	QE Version	No
3.14.0	10/03/15	Updated for x64 support	No

Acknowlegements

Frontpiece Design was produced by the chaoscope application.

Contents

1. Introduction 8
1.1 References 8
2. Single Threaded Run Time API Reference 9
2.1 Object Constructor 9
2.1.1 Parameters 9
2.1.2 Returns 9
2.1.3 Constraints 9
2.1.4 Usage 9
2.2 Database and Repository Functions 9
2.2.1 Open Database 9
2.2.2 Create Database 10
2.2.3 Create Replica 11
2.2.4 Delete Database 12
2.2.5 BuildIndex(es) 13
2.2.6 Get and Set Database Title 13
2.2.7 Read Database Icon Flags 14
2.2.8 Get Database Statistics 15
2.2.9 Get Database Information 15
2.2.10 ACL Operations 16
2.3 Application and Error Support Functions 16
2.3.1 Logging Functions 16
2.3.2 Error Handling 17
2.3.3 Memory Dump Functions 18
2.3.4 Program Execution 19
2.4 Convenience Functions 19
2.4.1 Formatting and Data Conversion Functions 20
2.4.2 Memory Functions 21
2.4.3 Comparison Functions 21
3. Multi-Threaded Run Time API Reference 23
3.1 Object Constructor 23
3.1.1 Parameters 23
3.1.2 Returns 23
3.1.3 Constraints 23
3.1.4 Usage 23
3.2 Asynchronous Execution Functions 24
3.2.1 PostARequest Function 24
3.2.2 GetRejoinRequest Function 25

3.2.3 IsRejoinRecommended Function....26
3.3 Constrained Multi Lane Scheduling Functions....27
3.3.1 GetThreadCount Function....27
3.3.2 GetCMLSRecommendation Function....27
3.4 Lightweight Thread Synchronisation Functions....28
3.4.1 AcquireAppMutex Function....28
3.4.2 FreeAppMutex Function....29
3.5 Thread Local Support Functions....29
3.5.1 GetMappedDBH Function....30
3.5.2 InvalidateNativeDBH Function....30
3.6 Miscellaneous Functions....31
3.6.1 ShowThreadStats Function....31
3.6.2 AttachCommandHandler Function....31
4. APIPackages Class....32
4.1 Object Constructor....32
4.2 Status Code Translation Functions....32
4.2.1 GetPackageID Function....32
5. CommandHandler Class....34
5.1 Object Constructor....34
5.1.1 Parameters....34
5.1.2 Returns....34
5.2 Handle Commands Function....34
5.2.1 Parameters....34
5.2.2 Usage....34
5.3 ParseCustomCommand Interface....35
5.3.1 Parameters....35
5.3.2 Returns....35
5.3.3 Usage....35
5.4 CustomCommandHandler Interface....35
5.4.1 Parameters....35
5.4.2 Returns....35
5.4.3 Usage....36
5.4.4 System Commands....36
5.4.5 System Commands for Debug Builds....38
5.5 GetAutoCommand Interface....38
5.5.1 Parameters....38
5.5.2 Returns....38
5.5.3 Usage....38
5.6 State Member....39

5.6.1 Usage....39
6. TransactionHandler Class....40
6.1 Object Constructor....40
6.1.1 Parameters....40
6.1.2 Returns....40
6.2 ProcessQueue Function....40
6.2.1 Parameters....40
6.2.2 Usage....41
6.3 MarshallTransaction Interface....41
6.3.1 Parameters....41
6.3.2 Returns....41
6.3.3 Usage....42
6.4 SerializeTransaction Interface....42
6.4.1 Parameters....42
6.4.2 Usage....42
6.5 Status Update Functions....43
6.5.1 Parameters....43
6.5.2 Usage....43
6.5.3 Transaction Queue Life Cycle....43
7. ElapsedTimer Class....45
7.1 Object Constructor....45
7.1.1 Returns....45
7.2 getElapsed Function....45
7.2.1 Returns....45
7.2.2 Usage....45
7.3 getElapsedMillis Function....46
7.3.1 Returns....46
7.3.2 Usage....46
8. Runnable Class....47
8.1 Object Constructor....47
8.2 ExecuteThisRequest Interface....47
8.2.1 Parameters....47
8.2.2 Usage....47
9. Debugging Classes....48
9.1 Helper Class....48
9.1.1 Object Constructor....48
9.1.2 ReportMemoryUsage Function....48
9.1.3 CreateMemoryDump Function....49
10. Non-Exposed Classes....51

10.1 ThreadDispatcher Class 51
10.2 ThreadManager Class 51
10.3 ThreadManagerPolicy Class 51
10.4 ThreadMonitor Class 51
10.5 ThreadScheduler Class 51
10.6 WorkerThread Class 51
11. Supporting Classes 52
11.1 RunSettings Class 52
11.1.1 Object Constructor 52
11.1.2 SetDefaults Interface 53
11.1.3 ValidateParameters Interface 53
11.1.4 ShowUsage Interface 53
11.1.5 ShowSettings Interface 54
11.1.6 AllowExecution Member 54
11.1.7 IsValid Member 54
11.1.8 EchoLog Member 54
11.1.9 NoRepository Member 55
11.1.10 NoAppLog Member 55
11.1.11 CreateRepository Member 55
11.1.12 RunningAsAddin Member 55
11.1.13 NeedsMQ Member 55
11.1.14 AllowMultipleAddins Member 56
11.1.15 LogLevel Member 56
11.1.16 TraceArea Member 56
11.1.17 szRepServer Member 57
11.1.18 szRepDb Member 57
11.1.19 APPName Member 57
11.1.20 APPTitle Member 58
11.1.21 APPVer Member 58
11.2 ThreadManagerPolicy Class 58
11.2.1 Object Constructor 58
11.2.2 TPSchedMode Member 58
11.2.3 TPoolPolicy Member 59
11.2.4 PriorityPolicy Member 59
11.2.5 TargetThreads Member 59
11.2.6 PendingRQECapacity Member 59
11.2.7 RejoinRQECapacity Member 60
11.2.8 AsyLogPoolEntries Member 60
11.2.9 MaxPctL0Threads Member 60

11.2.10 MaxPctL1Threads Member 60
11.2.11 MaxPctL2Threads Member 60
11.3 TransactionQueue Class 61
11.3.1 Object Constructor 61
11.3.2 wQueueProtocols Member 61
11.3.3 MaxConcurrent Member 62
11.3.4 MaxRunLimit Member 62
11.3.5 hdbQueue Member 62
11.3.6 MinDelay Member 62
11.3.7 ReQTXCycle Member 62
11.3.8 MaxReqCount Member 63
11.3.9 DelayCycleSecs Member 63
11.3.10 LocalPermit Member 63
11.3.11 QueueIsSuspended Member 63
11.3.12 szQueueName Member 63
11.3.13 szQServer Member 64
11.3.14 szQDbPath Member 64
11.3.15 szReadyQName Member 64
11.3.16 szInProgressQName Member 64
11.3.17 szDelayedQName Member 64
11.3.18 szSchedQName Member 65
12. Threading Model 66
12.1 Introduction 66
12.2 The Request Lifecycle 66
12.3 The Request Owner 68
12.4 Request Priority 68
12.5 Constrained Multi Lane Scheduling 68
12.6 The Design of Runnable Classes 69
12.7 Request Sizing 70

1. Introduction

The Domino eXplorer (DX) was developed as a means for facilitating the rapid development of tools to be used in projects that involve high volumes of data transformation in Notes Databases. DX has been, and continues to be developed for use across a wide range of Domino versions and platforms. The reference platforms are Domino 9.0.x on Windows Server 2003 R2 (32 & 64 bit) and Red Hat Linux 6.6. DX is also used as a research tool to investigate various aspects of Autonomic Systems, in particular Autonomic Throughput Optimisation.

Standardised utilities have also been built around some of the functional DX classes, these are published as “DX Tools” and can save time by providing off-the-shelf processing to be incorporated into complex transformations that need high throughput rates.

DX consists of a set of “Kernel” classes and a collection of “Functional” classes this document provides details of the Domino eXplorer (DX) Kernel API functions. DX provides a single-threaded or multi-threaded runtime environment for applications, the runtime exposes core functionality to applications through the Kernel API. The API for the “Functional” classes are documented in the “DX Tools: Class Catalogue” publication.

1.1 References

Title	Version	Date	Author
DX Tools Class Catalogue	3.12	17/12/2011	HMNL
DX Tools Application Design Guide	3.0	23/01/2012	HMNL

2. Single Threaded Run Time API Reference

This section contains the API specifications for all public members and functions that are exposed by the Single Threaded (ST) run time kernel object of the Domino eXplorer.

Header File: DXCommon/ExecEnvironment.h

2.1 Object Constructor

ExecEnvironment(`RunSettings *rsIn, int argc, char *argv[]`**)**

2.1.1 Parameters

Name	Type	Use
rsIn	RunSettings *	A pointer to a validated RunSettings object, or more usually an object of a class that extends the RunSettings class. The RunSettings class contains configuration data that determines how the Run Time is initialized. See “Supporting Objects” for more details.
argc	Int	Count of parameters passed to the main entry point of the application.
argv	char * []	Pointers to the array of parameters passed to the main entry point of the application

2.1.2 Returns

A pointer to the initialized ExecEnvironment object.

2.1.3 Constraints

None.

2.1.4 Usage

Creating a new Run Time object in your application will initialize the run time environment, including the underlying Notes/Domino run time. If a Repository Database is specified then this will be made available to the application and, according to the current settings logging will be initiated in the Repository.

2.2 Database and Repository Functions

2.2.1 Open Database

```
DBHANDLE OpenDatabase(char *szServer, char *szDatabase)

DBHANDLE OpenDatabase(char *szServer, char *szDatabase, int iThreadID)

DBHANDLE OpenDatabase(char *szServer, char *szDatabase, USHORT *usMode)

DBHANDLE OpenDatabase(char *szServer, char *szDatabase, USHORT *usMode, int
iThreadID)

DBHANDLE OpenDatabase(char *szServer, char *szDatabase, DBID *dbidDB)
```

```
DBHANDLE OpenDatabase(char *szServer, char *szDatabase, DBID *dbidDB, int iThreadID)

DBHANDLE OpenDatabase(char *szServer, char *szDatabase, WORD wOpenFlags, DBID *dbidDB, int iThreadID)
```

2.2.1.1 Parameters

Name	Type	Use
szServer	char *	Pointer to a null terminated character string that contains the name of the server on which you want to open the database. The name should be in either canonical or abbreviated format. An empty string or a string containing the literal value "Local" will cause the database to be opened on the local client or server.
szDatabase	char *	Pointer to a string containing the path and database name that is to be opened. The path is provided relative to the Notes Data directory. Alternatively the string can contain the ReplicaID of the database that is to be opened. The replica id should be provided in one of the following formats. XXXXXXXXXXXXXXXX XXXXXXXX:XXXXXXXX __XXXXXXXXXXXXXXXX.nsf
iThreadID	Int	For single threaded applications always specify 0 to indicate the main thread of a program.
usMode	USHORT *	The address of a USHORT that will be filled in with the Mode (Database or Directory) of the item that was opened.
dbidDB	DBID *	The address of a DBID that will be filled in with the DBID of the open database.
wOpenFlags	WORD	Flags that are passed to the Notes API NSFDbOpen call. See DBOPEN_xxx symbolic values in the Notes API reference.

2.2.1.2 Returns

A DBHANDLE to the open database or NULLHANDLE if the database could not be opened.

2.2.1.3 Constraints

None.

2.2.1.4 Usage

If the call returns a NULLHANDLE then the Notes API Return Code (STATUS) can be obtained by calling GetLastOpenError(). If the DBOPEN_FORCE_FIXUP flag is specified then the standard Notes API call will fail if the database is remote, OpenDatabase() will perform the fixup on a remote database and then open the database without the flag set.

2.2.2 Create Database

```
DBHANDLE CreateDatabase(char *szServer, char *szDatabase)

DBHANDLE CreateDatabase(char *szServer, char *szDatabase, int iThreadID)

DBHANDLE CreateDatabase(char *szServer, char *szDatabase, BOOL bTXLog, int iThreadID)
```

```
DBHANDLE CreateDatabase(char *szServer, char *szDatabase, BOOL bTXLog, DBID
*dbidDB, int iThreadID)
```

2.2.2.1 Parameters

Name	Type	Use
szServer	char *	Pointer to a null terminated character string that contains the name of the server on which you want to create the database. The name should be in either canonical or abbreviated format. An empty string or a string containing the literal value “Local” will cause the database to be created on the local client or server.
szDatabase	char *	Pointer to a string containing the path and database name that is to be created. The path is provided relative to the Notes Data directory.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.
bTXLog	BOOL	Specify TRUE to have transaction logging enabled on the new database or specify FALSE if transaction logging is not required.
dbidDB	DBID *	The address of a DBID that will be filled in with the DBID of the new database.

2.2.2.2 Returns

A DBHANDLE to the open database or NULLHANDLE if the database could not be created.

2.2.2.3 Constraints

If you are creating a database on a remote server then you will need the rights to be able to do this.

2.2.2.4 Usage

If the call succeeds then the database will have been initialised with a default ACL and a default view and design collection.

Default ACL

<*current user or server*> - Manager

-Default- - No Accedd

Anonymous - No Access

If the database is created on a server then the following additional entries are set.

<*target server*> - Manager (Admin Server)

LocalDomainServers - Manager

LocalDomainAdmins - Manager

If the call returns a NULLHANDLE then the Notes API Return Code (STATUS) can be obtained by calling GetLastOpenError().

2.2.3 Create Replica

```
DBHANDLE CreateReplica(char *szServer, char *szDatabase, DBHANDLE hdbSrc,
DBID *dbidDB)
```

```
DBHANDLE CreateReplica(char *szServer, char *szDatabase, DBHANDLE hdbSrc,
DBID *dbidDB, int iThreadID)

DBHANDLE CreateReplica(char *szServer, char *szDatabase, BOOL bTXLog,
DBHANDLE hdbSrc, DBID *dbidDB, int iThreadID)
```

2.2.3.1 Parameters

Name	Type	Use
szServer	char *	Pointer to a null terminated character string that contains the name of the server on which you want to create the database. The name should be in either canonical or abbreviated format. An empty string or a string containing the literal value “Local” will cause the database to be created on the local client or server.
szDatabase	char *	Pointer to a string containing the path and database name that is to be created. The path is provided relative to the Notes Data directory.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.
hdbSrc	DBHANDLE	Handle of the database that the new database will be a replica of.
bTXLog	BOOL	Specify TRUE to have transaction logging enabled on the new database or specify FALSE if transaction logging is not required.
dbidDB	DBID *	The address of a DBID that will be filled in with the DBID of the new replica.

2.2.3.2 Returns

A DBHANDLE to the open database or NULLHANDLE if the replica could not be opened.

2.2.3.3 Constraints

If you are creating a replica on a remote server then you will need the rights to be able to do this.

2.2.3.4 Usage

If the call returns a NULLHANDLE then the Notes API Return Code (STATUS) can be obtained by calling GetLastOpenError().

2.2.4 Delete Database

```
STATUS DeleteDatabase(char *szServer, char *szDatabase, BOOL bMakeOffline,
int iThreadID)
```

2.2.4.1 Parameters

Name	Type	Use
szServer	char *	Pointer to a null terminated character string that contains the name of the server from which you want to delete the database. The name should be in either canonical or abbreviated format. An empty string or a string containing the literal value “Local” will cause the database to be deleted from the local client or server.
szDatabase	char *	Pointer to a string containing the path and database name that is to be deleted. The path is provided relative to the Notes

		Data directory.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.
bMakeOffline	BOOL	Specify TRUE to force the database to be taken offline before it is deleted and FLASE if not.

2.2.4.2 Returns

The STATUS from the underlying Notes API calls.

2.2.4.3 Constraints

Specifying bMakeOffline as TRUE to force the database to be taken offline before it is deleted will only work for local (i.e. on the same domino instance where the application is running) databases.

2.2.4.4 Usage

Use the setting to force a database offline before deletion if you are running on a server that is using Transaction Logging.

2.2.5 BuildIndex(es)

```
BOOL BuildIndexes(void)
BOOL BuildIndexes(DBHANDLE hdbIX)
BOOL BuildIndexes(DBHANDLE hdbIX, int iThreadID)
void BuildIndex(DBHANDLE hdbIX, NOTEID nidView, int iThreadID)
```

2.2.5.1 Parameters

Name	Type	Use
hdbIX	DBHANDLE	Handle of the database in which the view index(as) are to be built.
nidView	NOTEID	The NOTEID of the view note that will have the view index built.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.2.5.2 Returns

BuildIndexes returns a BOOL with TRUE indicating that the view index rebuilds succeeded and FALSE indicating that the rebuilds could not be completed. **BuildIndex** does not return anything.

2.2.5.3 Constraints

None.

2.2.5.4 Usage

The first variant `BuildIndexes(void)` will rebuild all of the view indexes in the Repository Database, if one is in use.

2.2.6 Get and Set Database Title

```
void GetDbTitle(DBHANDLE hDB, char *szTitle, int iThreadID)
BOOL SetDbTitle(DBHANDLE hDB, char *szTitle)
BOOL SetRepositoryTitle(char *szTitle)
```

2.2.6.1 Parameters

Name	Type	Use
hDB	DBHANDLE	Handle of the database from which to get or set the title.
szTitle	char *	Pointer to a null terminated character string that contains the title to be set or in which the title will be returned.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.2.6.2 Returns

The two “Set” functions return a BOOL with TRUE indicating success and FALSE indicating that the call did not complete. The “Get” does not return anything but fills the buffer with the database title.

2.2.6.3 Constraints

None.

2.2.6.4 Usage

The **SetRepositoryTitle** function will set the title in the current Repository Database if one is in use.

2.2.7 Read Database Icon Flags

```
STATUS GetIconFlags(DBHANDLE hDB, char *szFlags, int iMaxLen)
STATUS GetIconFlags(DBHANDLE hDB, char *szFlags, int iMaxLen, int iThreadID)
BOOL IsIconFlagSet(DBHANDLE hDB, char cFlag)
BOOL IsIconFlagSet(DBHANDLE hDB, char cFlag, int iThreadID)
```

2.2.7.1 Parameters

Name	Type	Use
hDB	DBHANDLE	Handle of the database from which the flags will be read or tested.
szFlags	char *	Pointer to a buffer that will contain the returned flag array as a null terminated string.
iMaxLen	int	Length of the buffer to receive the flags
cFlag	char	Single character flag to be tested for.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.2.7.2 Returns

The **GetIconFlags** returns the STATUS from the underlying Notes API cal . **IsIconFlagSet** returns a BOOL, TRUE if the flag is set and FALSE if the flag is not set.

2.2.7.3 Constraints

None.

2.2.7.4 Usage

For the meaning of the individual flags see the CHFLAG_ area of stdnames.h in the Notes API.

2.2.8 Get Database Statistics

```
DWORD GetDataDocumentCount(DBHANDLE hDB, int iThreadID)
DWORD GetDbAllocatedSize(DBHANDLE hDB, int  iThreadID)
DWORD GetDbFreeSpace(DBHANDLE hDB, int  iThreadID)
```

2.2.8.1 Parameters

Name	Type	Use
hDB	DBHANDLE	Handle of the database from which the statistics will be retrieved.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.2.8.2 Returns

The **GetDataDocumentCount** returns the number of non-design documents in the database. The **GetDbAllocatedSize** returns the size of the database in 1 Kb (1024 byte) chunks. **GetDbFreeSpace** returns the amount of free space in the database in 1Kb (1024 byte) chunks. All functions return -1 if the statistic could not be determined.

2.2.8.3 Constraints

None.

2.2.8.4 Usage

2.2.9 Get Database Information

```
void GetReplicaID(DBHANDLE hDB, char *szRepID, int iLen, int iThreadID)
```

2.2.9.1 Parameters

Name	Type	Use
hDB	DBHANDLE	Handle of the database from which the information will be retrieved.
szRepID	char *	Pointer to a character buffer where the null terminated, decorated Replica ID will be returned.
iLen	int	Length of the buffer provided to hold the Replica ID.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.2.9.2 Returns

Nothing.

2.2.9.3 Constraints

None.

2.2.9.4 Usage

The decorated Replica ID is returned in the following format: `"%08lX:%08lX"`.

2.2.10 ACL Operations

```
BOOL GetACLRoles(HANDLE hACL, ACL_PRIVILEGES *pAllPrivs, char *szRoles, int iThreadID)
```

2.2.10.1 Parameters

Name	Type	Use
hACL	HANDLE	Handle of the ACL from which to derive the Roles.
pAllPrivs	ACL_PRIVILEGES	Pointer to an ACL_PRIVILEGES structure that will be populated with the Roles that are available in the database..
szRoles	char *	Pointer to a character buffer where all of the Role names will be placed.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.2.10.2 Returns

BOOL, TRUE if the call succeeded otherwise FALSE.

2.2.10.3 Constraints

None.

2.2.10.4 Usage

The function will format a string in the form “[<role name>][<role name>]” an entry will be provided for each possible Role (i.e. privilege bit), if a Role is not defined for a particular bit then the entry is returned as a pair of empty braces “[]”.. The bits in the ACL_PRIVILEGES structure will indicate which Roles are actually defined in the ACL.

2.3 Application and Error Support Functions

2.3.1 Logging Functions

```
BOOL LogMessage(char *szMsg)
BOOL LogMessage(char *szMsg, int iThreadID)
BOOL LogVerbose(char *szMsg)
BOOL LogVerbose(char *szMsg, int iThreadID)
BOOL LogTrace(int iTraceArea, char *szMsg)
BOOL LogTrace(int iTraceArea, char *szMsg, int iThreadID)
void LogSetVerbose(void)
void LogSetNormal(void)
void LogSetTrace(int iTraceArea)
void LogSetDebug(int iTraceArea)
void LogSetEchoOn(void)
void LogSetEchoOff(void)
int LogGetLevel(void)
BOOL LogIsEchoing(void)
```

```
void LogPushSettings(void)
void LogPopSettings(void)
void LogCommit(void)
```

2.3.1.1 Parameters

Name	Type	Use
szMsg	char *	Pointer to a null terminated string buffer containing the message that is to be written to the log.
iTraceArea	int	A reference to a particular area of the kernel that is associated with the current message.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.3.1.2 Returns

The **Logxxxx** functions that return a BOOL will return TRUE if the call succeeded and FALSE if problems were encountered during the logging operation. **LogGetLevel** returns the logging level that is currently in effect.

Logging Levels

LOGLEVEL_DEBUG	50	- Full debugging
LOGLEVEL_TRACE	40	- Include detailed functional tracing
LOGLEVEL_VERBOSE	30	- Include additional functional messages
LOGLEVEL_NORMAL	20	- Normal message level

LogIsEchoing returns an indication TRUE if the messages that are being written to the log are also being echoed to the console, FALSE if not.

2.3.1.3 Constraints

None.

2.3.1.4 Usage

LogVerbose and **LogTrace** functions will only write to the log if the current logging level is at or above the respective level. **LogVerbose** will only write log entries if the current level is VERBOSE, TRACE or DEBUG. **LogTrace** will only write log entries if the current level is TRACE or DEBUG, in addition **LogTrace** messages will only write log messages if the current trace area is the same as that supplied in the logging call or the current trace area is set to ANY (0).

The **LogPushSettings** and **LogPopSettings** will store an restore the current settings of the logging level and the trace area.

The **LogSetxxx** functions will set the current logging level and, if appropriate, the trace area.

The **LogCommit** function will force the current logging mechanism to write any buffered output.

2.3.2 Error Handling

```
void GetAPIMessage(STATUS stAPICode, char *szMsg)
void GetWINAPIError(DWORD dwError, char *szMsg)
void GetUNIXAPIError(int iError, char *szMsg)
void LogWINAPIError(void)
```

```
void LogWINAPIError(int iThreadID)
```

2.3.2.1 Parameters

Name	Type	Use
stAPICode	STATUS	A status code returned from a Notes API call.
szMsg	char *	Pointer to a string buffer where the formatted error message will be placed. The buffer should be at least MAX_MSG + 1 bytes in size.
dwError	DWORD	An error code that is set as the result of a call to a Windows API call, normally retrieved using the GetLastError() function.
iError	int	An error code that is set as the result of a UNIX API call
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.3.2.2 Returns

Nothing.

2.3.2.3 Constraints

None.

2.3.2.4 Usage

The **GetxxxMessage** functions take a Notes API, Windows API or a UNIX API error code and return a string that includes the formatted code and an explanation of the error. The LogWINAPIError functions will retrieve that last recorded error code and log a message that formats and explains the code.

2.3.3 Memory Dump Functions

```
void DumpMemory(void *lpMem, int iLen, char *szSymbol)
void DumpMemory(void *lpMem, int iLen, char *szSymbol, int iThreadID)
void DumpMemoryExt(void *lpMem, int iLen, char *szSymbol)
void DumpMemoryExt(void *lpMem, int iLen, char *szSymbol, int iThreadID)
```

2.3.3.1 Parameters

Name	Type	Use
lpMem	void *	A pointer to the area of memory to be dumped.
iLen	int	The length of the memory area to dump, in bytes.
szSymbol	char *	A null terminated string that is used to annotate the dump.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.3.3.2 Returns

Nothing.

2.3.3.3 Constraints

None.

2.3.3.4 Usage

The **DumpMemory** function will format a dump (hexadecimal and character) of the area of memory supplied by the pointer and of the supplied length and writes the dump output to the current log. The dump will only be written if the current logging level is DEBUG.

The **DumpMemoryExt** functions do the same as the DumpMemory functions but will produce the output whatever the current logging level is.

2.3.4 Program Execution

```
int OSLoadProgram(OSPROG_DEF *ospProg, int  iThreadID)
int OSLoadSrvCommand(OSPROG_DEF *ospProg, int  iThreadID)
```

2.3.4.1 Parameters

Name	Type	Use
ospProg	OSPROG_DEF *	A pointer to the structure that describes a program or a command that is to be executed
iThreadID	Int	For single threaded applications always specify 0 to indicate the main thread of a program.

```
typedef struct {
     char     szAppName[MAXPATH];         //  The name of the program to execute
     char     szWorkingDir[MAXPATH];      //  The working directory to be set for
execution
     char     szAppCmdLine[MAXPATH];      //  The command line to be passed to the
application
     DWORD    dwFlags;                    //  Flags controlling how the application is
run
     DWORD    dwReturnCode;               //  The exit code from the application
     DWORD    dwProcessId;                //  The process ID of the application
}OSPROG_DEF;
```

2.3.4.2 Returns

The return value is an integer code indicating if the program was run or started. A return code of zero indicates that the named program was successfully started or run.

2.3.4.3 Constraints

None.

2.3.4.4 Usage

Use the **OSLoadSrvCommand** variant of the call to run commands that are normally run as server commands, such as compact (ncompact.exe) etc.

2.4 Convenience Functions

The DX runtime provides a large number of convenience functions, these are documented in this section of the document. The functions are split into separate functional groups.

2.4.1 Formatting and Data Conversion Functions

```
BOOL GenUniqueID(char *szUniqueID, int iUniqueIDLen, int iThreadID)

void populateUNID(char *szUNID, UNID *uidFormatted)

void GetUNIDString(UNID unidSource, char *szTarget, int iTargetLen, BOOL bExtString,
int iThreadID)

void TrimToUpper_s(char *szDest, int iDest_Size, char *szSrc)

void RemoveEscapes(char *szMsg, int iStr_Size)

int xtoi(char *szHex)
```

2.4.1.1 Parameters

Name	Type	Use
szUniqueID	char *	Pointer to a buffer that will receive the generated Unique ID value.
iUniqueIDLen	Int	Size of the buffer that will receive the generated Unique ID value.
szUNID	char *	Pointer to a null terminated character string holding a formatted hexadecimal UNID value.
uidFormatted	UNID *	Pointer to a UNID structure that will be populated from the formatted string.
uidSource	UNID *	Pointer to a UNID structure that is to be formatted as a readable hexadecimal encoded character string.
szTarget	char *	Pointer to a buffer that will receive the formatted UNID.
iTargetLen	Int	Size of the buffer that will receive the formatted UNID.

bExtString	BOOL	Switch that determines the format of the returned UNID string.
		
		TRUE – Use decorated format
		`"OF: %008lX:%008lX - ON: %008lX:%008lX"`
		FALSE – Use undecorated format
		`"%008lX%008lX%008lX%008lX"`

szDest	char *	Pointer to a buffer that will receive the trimmed character string (null terminated).
iDest_Size	Int	Size of the buffer that will receive the trimmed string.
szSrc	char *	Pointer to a character string to be trimmed.
szHex	char *	
iThreadID	Int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.4.1.2 Returns

The GetUNiqueID function returns a BOOL TRUE if the function succeeded and FALSE if not.

The xtoi function returns an integer value of the hexadecimal string passed to the function.

2.4.1.3 Constraints

None noted.

2.4.1.4 Usage

The populateUNID and GetUNIDString functions are complementary, the first takes a character string containing a hexadecimal encoded UNID and will populate the UNID structure with the value. The second reverses the transformation, taking a UNID structure and converting it to a hexadecimal encoded string in either a decorated or undecorated format.

The xtoi function converts a string containing hexadecimal characters into an integer.

The TrimToUpper_s function will trim a character string, removing leading and trailing whitespace it will also fold all characters in the string to uppercase.

The RemoveEscapes function removes % characters from character strings, the function is used to make sure that messages that are destined for the Domino Console do not contain printf escape characters as these will cause errors in the console logging API.

The GenUnique function is a C API wrapper for the @Unique formula function.

2.4.2 Memory Functions

```
MEMHANDLE getDominoMemory(DWORD dwSize, int iThreadID)
```

```
MEMHANDLE resizeDominoMemory(MEMHANDLE hMem, DWORD dwSize, int iThreadID)
```

2.4.2.1 Parameters

Name	Type	Use
dwSize	DWORD	The size of a requested memory allocation or re-allocation.
hMem	MEMHANDLE	The handle of en existing memory allocation that is to re-allocated with a different size.
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program.

2.4.2.2 Returns

Both functions return a MEMHANDLE handle to the area of memory that has been allocated or re-allocated. The functions return NULLHANDLE if the function could not be completed.

2.4.2.3 Usage

The GetDominoMemory function allocates a new chunk of memory that is managed by Domino, the memory allocation is shared across all domino processes. The resizeDominoMemory function will change the size of an existing memory allocation.

2.4.3 Comparison Functions

```
BOOL IsThisADatabase(char *szFileSpec)
```

```
BOOL IsThisAReplicaID(char *szFileName, char *szFRID)
```

```
BOOL MatchesPattern(const char *szString, const char *szPattern)
```

```
BOOL MatchesPatterni(const char *szString, const char *szPattern)
```

2.4.3.1 Parameters

Name	Type	Use
szFileSpec	char *	Pointer to a null terminated character string containing an arbitrary file name.
szFileName	char *	Pointer to a null terminated character string containing an arbitrary file name.
szFRID	char *	Pointer to a buffer that will receive a normally formatted replica ID.
szString	char *	Pointer to a null terminated arbitrary character string.
szPattern	char *	Pointer to a null terminated arbitrary character string that can contain wildcard characters.

2.4.3.2 Returns

The IsThisADatabase function will return TRUE if the passed arbitrary file name conforms to the naming for a Notes Database, otherwise it will return FALSE.

The IsThisAReplicaID function will return TRUE if the passed arbitrary file name conforms to one of the allowed Replica ID patterns, the function will also populate a buffer passed by the caller with the Replica ID in standard format. The function returns false if the passed string is not a replica ID.

The MatchesPattern will return TRUE if the first parameter matches the wildcard string pattern passed as the second parameter, otherwise it returns FALSE.

The MatchesPatterni function performs the same match as the MatchesPattern function except that the match is case insensitive.

2.4.3.3 Usage

3. Multi-Threaded Run Time API Reference

This section contains the API specifications for all public members and functions that are exposed by the Multi-Threaded (MT) run time kernel object of the Domino eXplorer. As the MTExecutive class extends the Single Threaded run time class (ExecEnvironment) all of the public functions from that class are available in the multi-threaded run time.

Header File: DXCommon/MTX/MTExecutive.h

3.1 Object Constructor

```
MTExecutive(RunSettings *rsIn, int argc, char *argv[])
MTExecutive(RunSettings *rsIn, ThreadManagerPolicy *tmpIn, int argc, char *argv[])
```

3.1.1 Parameters

Name	Type	Use
rsIn	RunSettings *	A pointer to a validated RunSettings object, or more usually an object of a class that extends the RunSettings class. The RunSettings class contains configuration data that determines how the Run Time is initialized. See “Supporting Objects” for more details.
Argc	Int	Count of parameters passed to the main entry point of the application.
Argv	char * []	Pointers to the individual parameters passed to the main entry point of the application
tmpIn	ThreadManagerPolicy *	A pointer to a valid ThreadManagerPolicy object. The ThreadManagerPolicy class is used to determine several aspects of the behavior of the MT kernel. See “Supporting Classes” for more details.

3.1.2 Returns

A pointer to the initialized MTExecutive object.

3.1.3 Constraints

None.

3.1.4 Usage

Creating a new Run Time object in your application will initialize the run time environment, including the underlying Notes/Domino run time. If a Repository Database is specified then this will be made available to the application and, according to the current settings logging will be initiated in the Repository. A pool of worker threads will also be initialized for use by the application. If no ThreadManagerPolicy is provided during construction then the run time will be initialized with a default policy in effect. Refer to the section “Threading Model” for more details of how the multi-threading works in the MT run time.

3.2 Asynchronous Execution Functions

The following functions are used to send requests for execution in parallel and retrieve completed requests.

3.2.1 PostARequest Function

```
BOOL PostARequest(int iFlags, void *pOwner, void *pExecutor, void *pParms, UINT Attrs,
int iPriority, int iThreadID)
```

3.2.1.1 Parameters

Name	Type	Use
iFlags	int	An integer that contains a set of bit flags that determine how the passed request will be handled. Flags may be PXR_WAITIF_BUSY - If specified then the call is blocking and will wait until the request can be posted, otherwise it will return a failure if the request cannot be posted at the time of the call. PXR_APP_WAIT – If the call is blocking then this flag will determine how the wait is performed, if specified then the DX kernel AppWait function is used to sleep if not specified then the appropriate OS function is used to wait.
pOwner	void *	An arbitrary address that identifies the “Owner” of the current request, the owner address is used when polling for completed requests. This will often be set to the address of a higher level request.
pExecutor	void *	The address of an object that implements the “Runnable” interface, this is the object that contains the code that is capable of executing the request.
pParms	void *	The address of an arbitrary object that contains the parameter data for the request.
Attrs	UINT	A set of bit flags that indicate a profile of the current request, the characteristics defined by these flags are used to determine how the request will be dispatched. The flags may be a valid combination of the RQATTR_XXX symbolic values, see below.
iPriority	int	The relative base priority of this request, a higher value represents a higher priority and will affect the dispatch sequence of requests.
iThreadID	int	The identity of the thread that is posting this request, specify 0 if this is the main thread of the application.

The RQATTR_XXX symbolic values can be valid combinations of the following symbolic values, refer to the “Threading Model” for more details of how these attributes affect the operation of the core thread management functions.

- RQATTR_FIREANDFORGET – If set then no polling will be done for completion of this request. NOTE: the caller is responsilbe for disposing of the parameter data object for this request after it has been processed.
- RQATTR_REJOIN – This setting is the opposite of “FIRE AND FORGET”, the application will issue polling requests to retrieve this request once it has been completed.
- RQATTR_PRODUCER – This setting indicates that when this request is executed then it will generate more requests (in moderate numbers).
- RQATTR_MEGAPRODUCER – This setting indicates that when this request is executed then it will generate more requests (in large numbers).
- RQATTR_SERVICE – This setting indicates that the request will run as a service i.e. it will be long running, usually for the duration of the application execution and will only terminate as the result of an external signal or condition.
- RQATTR_CMLSL0 – If Constrained Multi Lane Scheduling is active then this request is for Lane 0 (Service Requests).
- RQATTR_CMLSL1 – If Constrained Multi Lane Scheduling is active then this request is for Lane 1 (Scavenger/Feeder Requests).
- RQATTR_CMLSL2 – If Constrained Multi Lane Scheduling is active then this request is for Lane 2 (Unit Of Work Requests).
- RQATTR_CMLSL3 – If Constrained Multi Lane Scheduling is active then this request is for Lane 3+ (Sub-Task Requests).

3.2.1.2 Returns

The function will return FALSE if either the parameters were invalid or the request could not be posted at this time and the call is in non-blocking mode (see the iFlags parameter). The function will return TRUE if the request was posted. Once the call has returned TRUE the kernel can be polled for completion.

3.2.1.3 Constraints

None.

3.2.1.4 Usage

Calls to PostARequest should be interspersed with calls to check for the completion of requests that have already been processed. The threading kernel has finite resources available for queuing requests for execution and storing requests after they have completed if these resources become exhausted then calls may block indefinitely.

3.2.2 GetRejoinRequest Function

```
int GetRejoinRequest(void *pOwner, void **pParms, int iThreadID)
```

3.2.2.1 Parameters

Name	Type	Use
pOwner	void *	An arbitrary address that identifies the “Owner” of any requests that will be checked for completion.
pParms	void **	The address of a pointer where the address of any parameter data objects will be returned for completed requests.
iThreadID	int	The identity of the thread that is posting this request, specify 0 if this is the main thread of the application.

3.2.2.2 Returns

The function will return an integer containing one of the following symbolic values.

- `RJR_RETURNED` – A completed request has been found for the specified owner, the ponter value is valid and may be used.
- `RJR_NONE_READY` – There were no completed requests for the specified owner, however, there are one or more requests waiting to be executed or currently executing.
- `RJR_NONE_EXIST` - There were no completed requests for the specified owner, and there are no requests waiting to be executed or currently executing.

3.2.2.3 Constraints

Calls to retrieve completed requests can be quite heavy and affect system throughput if over used. Use the IsRejoinRecommended function to determine if a GetRejoinRequest should be issued.

3.2.2.4 Usage

In a loop that is posting many requests there should be an inner loop that retrieves completed requests, this would normally loop while the GetRejoinRequest returns RJR_RETURNED. Once the posting loop has completed all of the requests should be drained from the request pools this would normally be done by looping until the GetRejoinRequest function returns RJR_NONE_EXIST. If the function returns RJR_NONE_READY in this loop then the loop should wait for an interval of time before polling again. Any information returned in completed requests can be accumulated, failed requests can be re-driven or appropriate error actions taken, the parameter data objects returned should of course be destroyed in these rejoin loops to prevent heap exhaustion.

The design pattern suggested above can, of course, be replaced by other designs determined by a particular applications needs and architecture. The application must take care however to avoid the exhaustion of the request handling resources by polling for completed requests at appropriate points in the application.

3.2.3 IsRejoinRecommended Function

```
BOOL IsRejoinRecommended(void *pOwner, int iThreadID)
```

3.2.3.1 Parameters

Name	Type	Use
pOwner	void *	An arbitrary address that identifies the “Owner” of any requests that will be checked for completion.
iThreadID	int	The identity of the thread that is making this call, specify 0 if this is the main thread of the application.

3.2.3.2 Returns

The function returns TRUE if the kernel has determined that it is appropriate for an application to poll for completed requests otherwise it returns FALSE.

3.2.3.3 Usage

Calls to retrieve completed requests can be quite heavy and affect system throughput if over used. Use the IsRejoinRecommended function to determine if a GetRejoinRequest should be issued. So prior to entering a loop to retrieve completed requests the application should use this function to determine if it is appropriate to do so at this time. The following pseudo-code illustrates the suggested usage.

```
if (IsRejoinRecommended(…))
      while (GetRejoinRequest(…) == RJR_RETURNED)
      {
            Process returned request.
      }
End if
```

3.3 Constrained Multi Lane Scheduling Functions

If Constrained Multi Lane Scheduling (CMLS) is active then the following functions provide additional capabilities for asynchronous request execution. Refer to the section “Threading Model” for more details of CMLS.

3.3.1 GetThreadCount Function

```
int GetThreadCount(void)
```

3.3.1.1 Returns

The function returns an integer value specifying the number of threads currently being used in the Thread Pool for executing asynchronous requests.

3.3.1.2 Usage

Refer to the section on the “Threading Model” for a discussion on the use of this function.

3.3.2 GetCMLSRecommendation Function

```
int GetCMLSRecommendation(int iLevel, int iThreadID)
```

3.3.2.1 Parameters

Name	Type	Use
iLevel	int	The CMLS Lane or level values use one of the RQATTR_CMLSXX symbolic values.
iThreadID	int	The identity of the thread that is making this call, specify 0 if this is the main thread of the application.

- RQATTR_CMLSL0 – If Constrained Multi Lane Scheduling is active then this request is for Lane 0 (Service Requests).
- RQATTR_CMLSL1 – If Constrained Multi Lane Scheduling is active then this request is for Lane 1 (Scavenger/Feeder Requests).
- RQATTR_CMLSL2 – If Constrained Multi Lane Scheduling is active then this request is for Lane 2 (Unit Of Work Requests).
- RQATTR_CMLSL3 – If Constrained Multi Lane Scheduling is active then this request is for Lane 3+ (Sub-Task Requests).

3.3.2.2 Returns

The GetCMLSRecommendation function will return an integer value the meaning of the value is given by the following symbolic values.

- CMLS_LANE_FULL – The requested CMLS lane is full, if an attempt is made to post a request for execution in this lane then the call will block until the congestion reduces.
- CMLS_LANE_AVAILABLE – There are resources available to post a request for execution in the specified lane. If a request is posted for this lane then it will be queued for execution.
- CMLS_LANE_AVAILABLE_NOW - There are resources available to post a request for execution in the specified lane. If a request is posted for this lane then it will be executed immediately.

3.3.2.3 Usage

Refer to the section on the “Threading Model” for a discussion on the use of this function.

3.4 Lightweight Thread Synchronisation Functions

The following functions provide the capability of synchronising processing between multiple threads. Refer to the section on the “Threading Model” for a discussion on why the “Lightweight” synchronisation functions are provided and how they can fail.

3.4.1 AcquireAppMutex Function

```
BOOL AcquireAppMutex(int iMutexID, int iThreadID)
```

3.4.1.1 Parameters

Name	Type	Use
iMutexID	int	An integer value that identifies the mutex that the caller wants control of. The values are specified with the APP_MUTEXT_1 through APP_MUTEX_10 symbolic values.
iThreadID	int	The identity of the thread that is making this call, specify 0 if this is the main thread of the application.

3.4.1.2 Returns

The function returns TRUE if the specified mutex is now controlled by the thread making the call and returns FALSE if another thread currently controls that mutex and the wait timeout limit has expired while waiting to gain control.

3.4.1.3 Usage

The application can use this function in a while loop to implement an infinite wait for the mutex to become free. Applications must free the mutex after the need to have exclusive control of the resource it represents is over, use the FreeAppMutex function to free the mutex.

3.4.2 FreeAppMutex Function

```
BOOL FreeAppMutex(int iMutexID, int iThreadID)
```

3.4.2.1 Parameters

Name	Type	Use
iMutexID	int	An integer value that identifies the mutex that the caller wants control of. The values are specified with the APP_MUTEXT_1 through APP_MUTEX_10 symbolic values.
iThreadID	int	The identity of the thread that is making this call, specify 0 if this is the main thread of the application.

3.4.2.2 Returns

The function returns TRUE if the specified mutex is now free to be acquired by other threads. It returns FALSE if there was an internal (Should Not Occur) failure that prevented release of the mutex.

3.4.2.3 Usage

Applications must free the mutex after the need to have exclusive control of the resource it represents is over, use the FreeAppMutex function to free the mutex.

3.5 Thread Local Support Functions

The following functions provide mapping functions to map global resources to local resources that may only be used by a particular thread. The function group currently only support Notes Database Handles and O/S file handles.

3.5.1 GetMappedDBH Function

```
DBHANDLE GetMappedDBH(DBHANDLE hdbNative, int iThreadID)
```

3.5.1.1 Parameters

Name	Type	Use
hdbNative	DBHANDLE	The DBHANDLE of an open database (returned by the first open call) that is to be mapped into the current thread.
iThreadID	int	The identity of the thread that is making this call, specify 0 if this is the main thread of the application.

3.5.1.2 Returns

The function will return a DBHANDLE that can be used by the current thread to access the database. The call will return NULLHANDLE if the call cannot be completed for instance if the native handle that was passed to the call is no longer open.

3.5.1.3 Usage

This call should be used in situations where a Notes Database is opened and will then be accessed by multiple requests that are being processed in multiple threads. The Notes API maintains locks at the Database Handle that only allow access from the same thread that opened the database, any attempt to access the database from another thread will result in a bad return code from the Notes API call. The first thread that opens a database should store the DBHANDLE returned and pass this in any sub-task requests that need to access the database. Processing of the sub-task requests should always use the mapped (i.e. thread local) DBHANDLE that is returned by calls to this function.

When the top level request has finished using the identified database it should make a call to the InvalidateNativeDBH function to signal to the kernel that any mapped (i.e. thread local) database handles can now be released. This should be done before closing the native database handle.

NOTE: For code compatibility the call can be used in a single threaded application, in this mode the call just returns the native DBHANDLE that is passed in the call.

3.5.2 InvalidateNativeDBH Function

```
BOOL InvalidateNativeDBH(DBHANDLE hdbNative, int iThreadID)
```

3.5.2.1 Parameters

Name	Type	Use
hdbNative	DBHANDLE	The DBHANDLE of an open database (returned by the first open call) that is to be identified as no longer in use by the thread local mapping functions.
iThreadID	int	The identity of the thread that is making this call, specify 0 if this is the main thread of the application.

3.5.2.2 Returns

The function returns TRUE if the DBHANDLE is now treated as no longer in use, it returns FALSE if theee call could not be completed.

3.5.2.3 Usage

When the top level request has finished using the identified database it should make a call to the InvalidateNativeDBH function to signal to the kernel that any mapped (i.e. thread local) database handles can now be released. This should be done before closing the native database handle.

NOTE: For code compatibility the call can be used in a single threaded application, in this mode the call does nothing.

3.6 Miscellaneous Functions

The following function provide additional support for multi-threaded applications.

3.6.1 ShowThreadStats Function

```
void ShowThreadStats()
```

3.6.1.1 Usage

A call to this function will cause detailed thread level statistics to be written to the application log.

3.6.2 AttachCommandHandler Function

```
void AttachCommandHandler(CommandHandler *chApp)
```

3.6.2.1 Parameters

Name	Type	Use
chApp	CommandHandler *	A pointer to a valid CommandHandler object. Pass NULL to detach the currently attached command handler.

3.6.2.2 Usage

Applications that are to be run as server add-in tasks will typically create a command handler by extending the DX CommandHandler class. Applications then use the AttachCommandHandler function to make the command handler active allowing it to asynchronously read commands from the application message queue (MQ) and respond to those commands with the appropriate actions. During application shutdown the code sequence should identify the point where MQ commands will no longer be processed and at that point they should deactivate the current command handler by making an AttachCommandHandler call passing NULL as the address of the command handler.

4. APIPackages Class

This class is constructed by the runtime and is used to translate a Notes API status code into a codified readable form. Applications will not normally access functions in this class directly instead they will use the GetAPIMessage() function in the runtime to obtain a standardised text line for any Notes API error code that is to be displayed or logged.

Header File: DXCommon/APIPackages.h

4.1 Object Constructor

```
APIPackages(void)
```

4.2 Status Code Translation Functions

The following functions are used to convert a STATUS code returned from a Notes API call into a readable format.

4.2.1 GetPackageID Function

```
void GetPackageID(STATUS stAPIRC, char *szPkgID, int iMaxLen)
```

4.2.1.1 Parameters

Name	Type	Use
stAPIRC	STATUS	The Notes API return code that is to be translated into a readable format.
szPkgID	char *	A pointer to a character buffer where the readable form of the status code will be returned.
iMaxLen	int	The size of the buffer where the readable form of the status code will be returned.

4.2.1.2 Usage

The call returns a readable string that contains the status code in the same format that it is found in the Notes API error string include files i.e. PKG_<package name>+<error number>.

For example.

A call to the function passing a status code of 0x0103 will return an package identifier of “PKG_OS+3”.

```
#define ERR_NOEXIST                 PKG_OS+3
      errortext(ERR_NOEXIST,        "File does not exist")
```

Applications will not normally access this function instead they will call the GetAPIMessage() function in the run time this returns a fully decorated message containing all of the information about the status code including the error text.

5. CommandHandler Class

An object of the CommandHandler class or more usually a class that extends it is constructed by the application and activated through a call to the runtime. The class provides default handling of the Message Queue (MQ) for Server Add-In tasks. Extending classes can implement additional commands and/or override, modify or extend the processing associated with the default command set.

Header File: DXCommon/MTX/CommandHandler.h

5.1 Object Constructor

```
CommandHandler(void)
CommandHandler(ExecEnvironment *xeParent, int iThreadID)
```

5.1.1 Parameters

Name	Type	Use
xeParent	ExecEnvironment *	Pointer to the current runtime object
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

5.1.2 Returns

The constructor returns a pointer to the newly created CommandHandler object.

5.2 Handle Commands Function

```
void HandleCommands(int iThreadID)
```

5.2.1 Parameters

Name	Type	Use
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

5.2.2 Usage

The handle commands function should be invoked regularly to check for the presence of commands on the Message Queue (MQ). Applications so not usually perform this task directly, instead they call the AttachCommandHandle function in the run time this delegates the regular calling of this function to the multi-threading kernel.

5.3 ParseCustomCommand Interface

```
WORD   virtual ParseCustomCommand(char *szCommand, char *szOptions, int iThreadID)
```

5.3.1 Parameters

Name	Type	Use
szCommand	char *	A pointer to a null terminated string containing a command returned from the Message Queue (MQ).
szOptions	char *	A pointer to a character buffer where any command options should be returned.
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

5.3.2 Returns

Any inheriting class that implements the interface should return a WORD containing a value that encodes for the command or the symbolic value MQ_COMMAND_INVALID if the command was not recognised by the custom handler. Any custom commands should be assigned unique values that start with MQ_COMMAND_USER+1 and upwards.

5.3.3 Usage

If the interface returns MQ_COMMAND_INVALID then the invalid command is reported to the console and the log and then ignored by the command handler. If the interface encodes a valid command value then that value and any command options that were detected are passed to the CustomCommandHandler interface for execution.

5.4 CustomCommandHandler Interface

```
WORD   virtual CustomCommandHandler(WORD wCommand, char *szOptions, int iThreadID)
```

5.4.1 Parameters

Name	Type	Use
wCommand	WORD	The encoded value of the command that is to be executed.
szOptions	char *	A pointer to a character buffer where any command options that were present on the command line will be passed.
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

5.4.2 Returns

Any inheriting class that implements the interface should return MQ_COMMAND_NULL if the passed command was executed and no further commands are to be executed. If the interface does not perform

any processing then it should return the value of the encoded command that was passed. After executing a command the interface can return the encoded value of any arbitrary command, in this case it can also set any command options in the buffer that was passed to the interface.

5.4.3 Usage

All commands including commands that are handled by default by the base command handler are passed to this interface allowing inheriting classes to modify the behaviour of behaviour of any system commands (see below) in addition to implementing additional commands. To override the processing provided by a system command the interface should detect the encoded system command perform the processing required and then return MQ_COMMAND_NULL. To extend a system command by performing additional processing the interface should detect the command and then perform the additional processing and return the original passed command encoding.

5.4.4 System Commands

The following system commands are implemented by the base command handler.

5.4.4.1 Quit – MQ_COMMAND_QUIT

The quit command is not normally used, however, during a server shutdown the server issues this command on all message queues. An application will respond to the command by shutting down.

5.4.4.2 Stop [now] – MQ_COMMAND_STOP

The stop command is used to shut down an application in an orderly manner. Any transactions that are currently running will be completed, no new transactions are dispatched and the Server Add-In will shut down. Specifying the optional “now” parameter on the stop command causes an application to fail any transactions that are currently running and then shut down in an orderly manner.

5.4.4.3 Abort – MQ_COMMAND_ABORT

The abort command is an alias for the “stop now” command.

5.4.4.4 Suspend – MQ_COMMAND_SUSPEND

The suspend command tells an application to stop executing new transactions from the ready queue. Any transactions that are currently executing are completed, the Add-In task continues to run but will not process any transactions until the “resume” command is executed.

5.4.4.5 Resume – MQ_COMMAND_RESUME

The resume command is the antithesis of the suspend command. The command only has any effect if the Add-In task is in the suspended state, then it causes the processor to resume processing transactions from the ready queue.

5.4.4.6 Verbose – MQ_COMMAND_VERBOSE

The verbose command causes the logging mode of the processor to be switched to verbose mode, in this mode more detailed logging is made to the application log.

5.4.4.7 Loud – MQ_COMMAND_LOUD

The loud command is an alias for the verbose command.

5.4.4.8 Terse – MQ_COMMAND_TERSE

The terse command switches the logging mode of the processor back to normal mode, in this mode minimal logging is done to the application log.

5.4.4.9 Quiet – MQ_COMMAND_QUIET

The quiet command is an alias for the terse command.

5.4.4.10 Echo [on|off] – MQ_COMMAND_ECHO

The echo command without any parameters is the same as the “echo on” command it will cause all current application logging to be echoed to the Domino Server console and therefore the Domino log. The “echo off” command turns off the echo of application logging.

5.4.4.11 Noecho – MQ_COMMAND_NOECHO

The noecho command is an alias for “echo off”.

5.4.4.12 Trace [nnn] – MQ_COMMAND_TRACE

The trace command sets the processor logging functions into trace mode. The number on the command designates a particular are of function to be traced. Refer to the DXGlobals.h header file for the different trace area specifications.

This command should only be used for problem diagnosis.

In this mode very detailed logging is produced in the application log.

5.4.4.13 Debug [nnn] – MQ_COMMAND_DEBUG

The debug command sets the processor logging functions into debug mode. The number on the command designates a particular are of function to be traced. Refer to the DXGlobals.h header file for the different trace area specifications.

This command should only be used for problem diagnosis.

In this mode even more detailed logging is produced in the application log.

5.4.4.14 Refresh – MQ_COMMAND_REFRESH

The refresh command causes the an application processor to finish processing any transactions that are currently processing and then reset the processing environment to the default configuration and resume processing transactions.

5.4.4.15 Status – MQ_COMMAND_STATUS

The status command causes the processor to display the current state of the processor and some volumetric information about how many transactions have been processed.

Sample output:

```
01/06/2011 14:17:11 CET: DXR0907I: Command received: status. [500]

01/06/2011 14:17:11 CET: QCP0201I: 1 transaction have been dispatched, 0
completed, 1 are running, max concurrency is 1. [500]

01/06/2011 14:17:11 CET: QCP0208I: Transactions marked Completed: 0, Error:
0, Retried: 0, Delayed: 0. [500]
```

5.4.4.16 Stats [thread|debug] – MQ_COMMAND_STATS

The stats command causes a number of current values of statistics from the DXCommon kernel to be written to the applications log. The thread parameter on the command adds certain additional “per thread” statistics to be output. The debug operand on the command causes the “per thread” statistics to be included along with more details. Understanding these statistics is beyond the scope of this document, refer to the documents about the architecture of the DXCommon kernel to gain insight into the meaning of these statistics.

5.4.4.17 Panic [message] – MQ_COMMAND_PANIC

The panic command will trigger an NSD exception from within the processor. This is an extreme diagnostic aid as it will cause an NSD of the entire server. The optional message is recorded in the log and in the memory displayed in the NSD dump.

5.4.5 System Commands for Debug Builds

The following additional commands are implemented in debug builds of an application.

NOTE: The application must set the address of the debug Helper object in the command handler for these commands to be available.

5.4.5.1 Memory – MQ_COMMAND_MEMORY

The memory command shows a report on current memory usage by the application, these statistics are reported to the server console and the application log.

5.4.5.2 Dump – MQ_COMMAND_DUMP

The dump command causes the processor to generate a Windows Core Minidump of the application. The application continues to execute so the command can be issued a number of times during an execution of the application. The contents of the dump can be investigated using the standard Windows debugging tools (e..g. windbg).

5.5 GetAutoCommand Interface

```
WORD   virtual GetAutoCommand(char *szOptions, int iThreadID)
```

5.5.1 Parameters

Name	Type	Use
szOptions	char *	A pointer to a character buffer where any command options will be passed.
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

5.5.2 Returns

Any inheriting class that implements the interface should return a WORD containing a value that encodes for the command. Any custom commands should be assigned unique values that start with MQ_COMMAND_USER+1 and upwards.

5.5.3 Usage

This interface is called periodically to determine if a command should be “injected” into the command stream as if it had been entered through the console as a Tell command. The interface can be used to introduce simulated console commands on a timed basis or in response to other events from within the application.

5.6 State Member

```
int    State
```

5.6.1 Usage

The State member of the command handler is used to publish the current state of the object to implementing applications. Applications are expected to monitor the State member and respond to changes in the state with appropriate behaviour, the monitoring of the command handler state is normally done within the main routine of a server add-in task. The member consists of a number of bit flags that can be combined to show a particular state, the following symbolic values are used to interrogate the state.

- CH_STATE_QUIT – A “quit” signal has been detected by the command handler, applications should respond by shutting down immediately in response to this signal.
- CH_STATE_STOP – A “stop” command has been received from the command stream, applications should shut down in an orderly manner in response to this signal. Normally applications would be expected to finish processing any transactions that are currently running and then perform a controlled shutdown. Applications should not initiate the processing of any new transactions after detecting the stop signal.
- CH_STATE_ABORT – An “abort” or “stop now” command has been received from the command stream, applications should shut down as quickly as possible in a controlled manner. Any running transactions should be failed but clean-up processing may be executed.
- CH_STATE_SUSPEND – A “suspend” command has been received from the command stream, applications should respond by inhibiting the initiation of any new transactions until this signal is revoked (by issuing a “resume” command). Applications should continue to monitor the state to see if the signal is revoked or one of the shut down signals is asserted.
- CH_STATE_REFRESH – A “refresh” command has been received from the command stream, applications should perform whatever processing is needed to load or re-instate the current execution configuration and then clear the signal. This mechanism allows applications to be dynamically reconfigured in response to configuration changes without the need to shut down and restart the applications.
- CH_STATE_TERMINAL – The other state bits set are now regarded as permanent they will not change and applications should respond appropriately to the other state signals present.
- CH_STATE_INVALID – The command handler has detected an internal or external condition that implies that the command handler has become invalid and cannot be relied on to signal other states. Applications detecting this signal should shut down with appropriate error messages.

6. TransactionHandler Class

This class must be extended to provide a default implementation for handling the reading, dispatch and status recording for a queue of transactions. A transaction handler object and the associated transaction queue would be created by the application and then dispatched for asynchronous execution. Once dispatched the object will monitor the associated transaction queue and respond by dispatching any transactions that appear in the queue. The handler recognises a structured set of sub-queues that allow for the automated retry of failed transactions and the repeated execution of transaction on a fixed schedule.

Header File: DXCommon/MTX/CommandHandler.h

6.1 Object Constructor

```
TransactionHandler(void)
TransactionHandler(MTExecutive *xeParent, int iThreadID)
```

6.1.1 Parameters

Name	Type	Use
xeParent	ExecEnvironment *	Pointer to the current runtime object
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

6.1.2 Returns

The constructor returns a pointer to the newly created TransactionHandler object.

6.2 ProcessQueue Function

```
void ProcessQueue(TransactionQueue *tqNew, int iThreadID)
```

6.2.1 Parameters

Name	Type	Use
tqNew	TransactionQueue *	Pointer to a valid TransactionQueue object that describes the queue to be processed by the handle.
iThreadID	int	Specify the number of the thread that is making the call. Specify 0 for the main thread of the application.

6.2.2 Usage

This function is not normally used by applications it will synchronously process transactions in the defined transaction queue and will block until a state change is signalled to the queue telling it to stop. In normal usage the processing of a transaction queue is done asynchronously this is accomplished by posting a transaction queue object to be executed by a transaction handler. Refer to the section on “supporting objects” for details of the TransactionQueue Class.

6.3 MarshallTransaction Interface

```
int virtual MarshallTransaction(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, void **txObject, void **xxObject, UINT *Attrs, int *Priority, int
iThreadID)
```

6.3.1 Parameters

Name	Type	Use
nidTX	NOTEID	The NoteID of the document that will be marshalled into an object containing the transaction information.
hnTX	NOTEHANDLE	Handle of the transaction document that is to be marshalled into an object containing the transaction information.
tqCurrent	TransactionQueue *	A pointer to the TransactionQueue object that represents the queue from which the transaction was loaded.
txObject	void **	A pointer to a variable in which the interface will place the address of the object created to contain the transaction information.
xxObject	void **	A pointer to a variable where the interface will place the address of the object that will be used to execute the transaction.
Attrs	UINT *	The address of a variable where the interface will set any asynchronous posting attributes for this transaction. Refer to the RQATTR_XXXX symbolic definitions for possible values.
Priority	int *	The address of an integer variable where the interface will set the initial priority for asynchronous posting of the transaction.
iThreadID	int	Specify the number of the thread that is making the call. Specify 0 for the main thread of the application.

6.3.2 Returns

The interface must return an integer value that indicates what should be done with the transaction. The following symbolic values are defined to determine the processing disposition.

- `TX_DISPOSITION_NORMAL` – The transaction should be executed and the transaction document should be arked as “In Progress”.

- `TX_DISPOSITION_ERROR` – Do not execute the transaction instead flag the transaction document as an “Error”, the path of processing depends on the error retry configuration of the transaction and transaction queue.
- `TX_DISPOSITION_DELAY` - Do not excute the transaction move it to the delayed transaction queue for later execution.
- `TX_DISPOSITION_PANIC` – Do not execute the transaction mark the transaction document as an “Error” and prevent any further processing of the transaction queue.

6.3.3 Usage

This interface is used to convert a transaction from the serialised on-disk form into the in-memory object that will be processed.

The “Delay” mechanism has a number of possible uses from restricting particular transactions to executing during particular time windows to checking the usage levels on a server that is a target of a transaction and delaying it if the server is too busy at the present time.

Transactions that are managed by the Transaction Handler mechanisms must be capable of asynchronous execution.

6.4 SerializeTransaction Interface

```
void virtual SerializeTransaction(void *txObject, DBHANDLE hdbQueue, TransactionQueue
*tqCurrent, int iThreadID)
```

6.4.1 Parameters

Name	Type	Use
txObject	void *	A pointer to the object containing the transaction information.
hdbQueue	DBHANDLE	The handle of the database that is associated with the passed transaction queue.
tqCurrent	TransactionQueue *	A pointer to the TransactionQueue object that represents the queue from which the transaction was loaded.
iThreadID	int	Specify the number of the thread that is making the call. Specify 0 for the main thread of the application.

6.4.2 Usage

Inheriting classes must implement this interface. As a minimum implementation it must update the document for the transaction with an appropriate status code and destroy (delete) the passed transaction object. Usual implementation will also write processing details, for example statistics and logs, to the transaction document.

For changing the status of transactions there are a standard set of functions that can be used or overridden.

6.5 Status Update Functions

```
void virtual MarkTransactionDelayed(NOTEID nidTX, NOTEHANDLE hnTX, BOOL bInProgress,
TransactionQueue *tqCurrent, int iThreadID)

void virtual MarkTransactionReady(NOTEID nidTX, NOTEHANDLE hnTX, BOOL bInProgress,
TransactionQueue *tqCurrent, int iThreadID)

void virtual MarkTransactionError(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)

void virtual MarkTransactionInProgress(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)

void virtual MarkTransactionCompleted(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)

void virtual MarkTransactionRetried(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)
```

6.5.1 Parameters

Name	Type	Use
nidTX	NOTEID	The NoteID of the document that will be marshalled into an object containing the transaction information.
hnTX	NOTEHANDLE	Handle of the transaction document that is to be marshalled into an object containing the transaction information.
bInProgress	BOOL	A flag if the value is TRUE then the transaction came from the “In Progress” (currently executing queue).
tqCurrent	TransactionQueue *	A pointer to the TransactionQueue object that represents the queue from which the transaction was loaded.
iThreadID	int	Specify the number of the thread that is making the call. Specify 0 for the main thread of the application.

6.5.2 Usage

There are is a default implementation for each of the status update functions.

6.5.3 Transaction Queue Life Cycle

The following describes the normal transaction life cycles managed by the transaction handler.

Transactions may be marked for once-off execution at a particular point in time or for repeated execution at particular time intervals. These transactions have a status value of “SCHEDULED” and reside on the Schedule Queue. When the execution time is reached or the scheduled interval expires then these transactions are copied to the Ready Queue.

The Ready Queue is the queue monitored by the transaction handle to find work that is ready to be executed. These transactions have a status value of “NEW”. When transactions are marshalled for execution they are moved to the In Progress Queue. If the transaction should not be processed at the current time, for whatever reason then it is marked with a status value of “DELAYED” and moved to the Delayed Queue.

Transactions on the In Progress Queue are considered to be currently executing and have a status value of “INPROGRESS”.

Once a transaction has completed processing it will be passed to the SerializeTransaction interface to determine the disposition and move it to the appropriate queue. If the transaction completed successfully it will be stamped with a status value of “COMPLETED” and moved to the Completed Queue. If the transaction failed then the destination is determined by the transaction retry settings for the current queue and transaction. If the transaction can be retried and the retry limit has not been exhausted then a new copy of the transaction is marked with the “DELAYED” status value and moved to the Delayed Queue, the original transaction is stamped with the “RETRIED” status value and moved to the Delayed Queue for later execution. If the transaction has failed and does not support retries or the retry limit has been exhausted then it will be marked with the “ERROR” status value and moved to the Error Queue.

The Delayed Queue is scanned occasionally and if a transaction has been on that queue for long enough then it is marked with the “NEW” status value and returned to the Ready Queue.

When a transaction handler starts to process a queue then, depending on settings, it may scan the In Progress Queue and process any transactions as if they had failed.

7. ElapsedTimer Class

Objects of the ElapsedTime class provide a standard means of establishing elapsed (i.e. wall clock) time between events.

The class provides a getElapsed() method that returns a clock_t value containing the number of elapsed “ticks” since the object was created. A “tick” is platform dependent, the CLOCKS_PER_SEC defined symbol provides the number of “ticks” in a second on the target platform.

For longer intervals the class provides the getElapsedMillis() method that returns a clock_t value containing the number of elapsed milliseconds since the object was created.

The runtime provides a default ElapsedTimer object that is created during initialisation of the runtime, this object can be accessed through the “RunningTime” member of the runtime object.

Header File: DXCommon/ElapsedTimer.h

7.1 Object Constructor

```
ElapsedTimer(void)
```

7.1.1 Returns

A pointer to the newly constructed ElapsedTimer object.

7.2 getElapsed Function

```
clock_t getElapsed(void)
```

7.2.1 Returns

A clock_t containing the number of elapsed clock “ticks” since the object was created. A “tick” is platform dependent, the CLOCKS_PER_SEC defined symbol provides the number of “ticks” in a second on the target platform.

7.2.2 Usage

The number of CLOCKS_PER_SEC can be large on some platforms so this function should only be used for measuring shorter elapsed intervals, up to 10 minutes, for longer intervals use the getElapsedMillis function. Typical usage is to record the elapsed time at the start of an interval and again at the end of an interval and then subtract the first measurement from the second to determine the duration of the interval in ticks, divide the result by CLOCKS_PER_SEC to convert the result to seconds or by CLOCKS_PER_SEC/1000 to convert the result to milliseconds.

7.3 getElapsedMillis Function

```
clock_t getElapsedMillis(void)
```

7.3.1 Returns

A clock_t containing the number of elapsed milliseconds since the object was created.

7.3.2 Usage

Typical usage is to record the elapsed time at the start of an interval and again at the end of an interval and then subtract the first measurement from the second to determine the duration of the interval in ticks, divide the result by 1000 to convert to seconds.

8. Runnable Class

The Runnable abstract class defines an interface for any class that is to be dispatchable by the multi-threaded runtime.

Header File: DXCommon/MTX/Runnable.h

8.1 Object Constructor

```
Runnable(void)
```

8.2 ExecuteThisRequest Interface

```
void virtual ExecuteThisRequest(void * pReqObject, int iThreadID)
```

8.2.1 Parameters

Name	Type	Use
pReqObject	void *	A pointer to the object containing the transaction information.
iThreadID	int	The identiy of the thread that is being invoked with this request.

8.2.2 Usage

Implementing classes should cast the passed request object to the correct type and then vector the call to the appropriate function according to the content of the request.

9. Debugging Classes

9.1 Helper Class

This class is only implemented for Windows Platforms and only exposes functionality in Debug builds.

The Helper class is used to establish an object that provides additional diagnostic capabilities to the runtime environment. Objects of this class should only be constructed in DEBUG build configurations of an application.

The class provides services for monitoring application memory usage and producing Core Dumps on demand.

Header File: DXCommon/Debug/Helper.h

9.1.1 Object Constructor

```
Helper(ExecEnvironment *xeParent, int iThreadID)
```

9.1.1.1 Parameters

Name	Type	Use
xeParent	ExecEnvironment *	Pointer to the current runtime object
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

9.1.1.2 Returns

A pointer to the newly created Helper object.

9.1.2 ReportMemoryUsage Function

```
void ReportMemoryUsage(BOOL bReset, int iThreadID)
void ReportMemoryUsage(BOOL bReset, BOOL bEcho, int iThreadID)
```

9.1.2.1 Parameters

Name	Type	Use
bReset	BOOL	Set to TRUE to cause the initial values of the memory statistics to be reset to the current values (after they are reported). Set to FALSE to continue reporting against the initial values recorded when the object was constructed.
bEcho	BOOL	A switch when TRUE then the report is echoed to the console.
iThreadID	int	Specify the number of the thread that is creating the object. Specify 0 for the main thread of the application.

9.1.2.2 Usage

Calls to the ReportMemoryUsage generate a report in the current log and optionally on the console of the current allocation of memory by the application, the call also reports on the difference in memory allocation since last reported and since the Helper was created.

The functions report on the memory usage by the application in the C runtime heap.

Sample Output:

Current(3) Working Set size: 30704 Kb, +328 Kb since last measured, +4580 Kb since first measured, Peak: 30704 Kb.

Current(3) Paged Pool use: 1252 Kb, 0 Kb since last measured, 0 Kb since first measured, Peak: 1252 Kb.

Current(3) Non-Paged Pool use: 10 Kb, 0 Kb since last measured, 0 Kb since first measured, Peak: 10 Kb.

Current(3) Normal Objects on the Heap: 28 , 0 since last measured, +7 since first measured, Peak: 28.

Current(3) Normal Objects Allocation: 124 Kb, 0 Kb since last measured, +2 Kb since first measured, Peak: 124 Kb.

Current(3) Client Objects on the Heap: 0 , 0 since last measured, 0 since first measured, Peak: 0.

Current(3) Client Objects Allocation: 0 Kb, 0 Kb since last measured, 0 Kb since first measured, Peak: 0 Kb.

The report indicates the sequence number of memory reports “Current(3)” indicates the third time that the reporting method has been called. Each line of output references a different memory statistic and shows the current, delta since last reported, delta since first reported and the peak measurement of the particular statistic. The statistics of particular focus for programmers are the count and size of “Normal Objects” on the heap, steady increases in these values would indicate a leak of C++ objects from within the application.

It should be noted that Debug compilations of DX applications also enables the C runtime memory leak tracing protocols. At any point in a program a call can be made to the C Runtime “CheckMemoryLeaks()” method and this will report on each object that is allocated on the Heap giving the source file and line number where it was allocated as well as the size of the object. A call to CheckMemoryLeaks should be made immediately before an application terminates this will show any objects that remain allocated and provides an excellent means of detecting and fixing leaks caused by failing to delete C++ objects or failing to free memory allocations.

9.1.3 CreateMemoryDump Function

```
void CreateMemoryDump(int iThreadID)
```

9.1.3.1 Parameters

Name	Type	Use
iThreadID	int	For single threaded applications always specify 0 to indicate the main thread of a program otherwise specify the number of the thread that is making the call.

9.1.3.2 Usage

The CreateMemoryDump function can be called at any point in an application to create a “minidump” file of the application process including heap memory. The memory dump files are created in the “IBM_TECHNICAL_SUPPORT” sub-directory in the Notes data directory. The minidump files can be loaded into Visual Studio for contextual analysis or in the Windows Debugger (WinDbg).

The dump files are created with a standard name format:

DXDump-<appname>YYYYMMDD-HHMMSS.dmp.

Where <appname> is the name of the application and YYYYMMDD-HHMMSS is the timestamp that the dump was created.

10. Non-Exposed Classes

The following classes do not expose any functionality directly to applications.

10.1 ThreadDispatcher Class

A singleton object of this class forms part of the multi-threaded runtime system, it has the responsibility for dispatching units of work that are ready to be executed into an available worker thread.

Header File: DXCommon/Threads/threadDispatcher.h

10.2 ThreadManager Class

A singleton object of this class provides the master component of the multi-threaded runtime system .

Header File: DXCommon/Threads/threadManager.h

10.3 ThreadManagerPolicy Class

An object of this class contains information used by the thread manager to configure the multi-threaded runtime system. An application can configure an object of this class and use it in the creation of the runtime system to influence many settings and constraints that are used by the runtime system.

Header File: DXCommon/Threads/ThreadmanagerPolicy.h

10.4 ThreadMonitor Class

A singleton object of this class forms part of the multi-threaded runtime system, it has the responsibility for monitoring several aspects of the multi-threaded runtime system this includes such housekeeping actions as writing log entries.

Header File: DXCommon/Threads/threadMonitor.h

10.5 ThreadScheduler Class

A singleton object of this class forms part of the multi-threaded runtime system, it has the responsibility for maintaining the relative priority of units of work that are waiting to be executed.

Header File: DXCommon/Threads/threadScheduler.h

10.6 WorkerThread Class

This class provides objects that manage the execution of units of work in a single thread.

Header File: DXCommon/Threads/WorkerThread.h

11. Supporting Classes

The following classes are used to create data objects that communicate information between the application and the DX kernel components.

11.1 RunSettings Class

An object of the RunSettings abstract class or more usually a class that extends the RunSettings class is used to contain application and run specific information that conditions the configuration of the runtime and the application. Static application wide configuration data is set in the object along with parameters passed on the command line, these are then used to control the configuration of the runtime and the application.

For example the name, version and short description of the application are used at various places in the runtime, these are accessed through the RunSettings object that is used to initialise the runtime. The runtime also has a special database called the “Repository” (this database is optional for the runtime) it can be used for different purposes such as the destination for persistent logging. Typically the server and file name for the repository would be supplied as command line parameters and then set in the RunSettings object, if set the repository database will be opened during initialisation of the runtime.

The setting of the members in the RunSettings class is the responsibility of the implementing class.

Header File: DXCommon/RunSettings.h

11.1.1 Object Constructor

```
RunSettings(void)
RunSettings(int argc, char * argv[])
```

11.1.1.1 Parameters

Name	Type	Use
argc	int	The count of arguments that were passed to the main entry point of the program.
argv	char * []	The array of parameters that were passed to the main entry point of the application.

11.1.1.2 Implementation Pattern

Inheriting classes should implement the following pattern for the constructor to ensure correct interfacing with the run time.

```
//  Constructor
if (SetDefaults())            //  Set the runtime defaults
{
      IsValid = ValidateParameters(argc, argv); //  Validate and capture the
parameters
}
if (IsValid)
{
      if (LogLevel > LOGLEVEL_NORMAL)
      {
```

```
            //  Show the current parameter settings
            ShowSettings();
        }
}
```

11.1.2 SetDefaults Interface

```
BOOL virtual SetDefaults(void)
```

11.1.2.1 Returns

The implementation should return TRUE if the defaults were set correctly otherwise return FALSE.

11.1.2.2 Usage

The SetDefaults interface must be implemented in the inheriting class. The defined method should be invoked in the constructor of the implementing class. The implementation should ensure that all default member values are set in both the base class and the inheriting class.

11.1.3 ValidateParameters Interface

```
BOOL virtual ValidateParameters(int argc, char* argv[])
```

11.1.3.1 Parameters

Name	Type	Use
Argc	int	The count of arguments that were passed to the main entry point of the program.
Argv	char * []	The array of parameters that were passed to the main entry point of the application.

11.1.3.2 Usage

The ValidateParameters interface must be implemented in the inheriting class. The defined method should be invoked in the constructor of the implementing class. The implementation should ensure that all default member values are set in both the base class and the inheriting class from any parameters supplied on the command line.

11.1.4 ShowUsage Interface

```
void virtual ShowUsage(void)
```

11.1.4.1 Usage

The ShowUsage interface should be implemented in inheriting classes. The implementation should display messages on the standard output device (STDOUT) to show the correct usage of the application.

11.1.5 ShowSettings Interface

```
void virtual ShowSettings(void)
```

11.1.5.1 Usage

The ShowUsage interface should be implemented in inheriting classes. The implementation should display messages on the standard output device (STDOUT) to show the current settings that are in effect for this run of the application.

11.1.6 AllowExecution Member

```
BOOL   AllowExecution
```

Set this member to TRUE if the application should be allowed to continue execution, otherwise set the value to FALSE. This member should be checked in the main routine of the invoking application to determine if it is safe to continue program execution with the current settings in effect.

11.1.7 IsValid Member

```
BOOL   IsValid
```

Set this value to TRUE by default and set it to FALSE in the event of any failure to parse any of the current application run parameters.

11.1.8 EchoLog Member

```
BOOL   EchoLog
```

Set this member to TRUE if all logging messages are to be echoed to the console and set FALSE if not. If the application is running on a server then messages will be echoed to the Domino Server Console, if running on a workstation then messages are echoed to the command window from which the application was invoked.

11.1.9 NoRepository Member

BOOL NoRepository

Set this member to TRUE to prevent the kernel from using a repository database, otherwise set it to TRUE. The repository database is used by the kernel as a default destination for logging and a default source for transactions. The opening and closing of the repository database, if used, is intrinsic to the kernel. Applications can obtain a handle to the repository database for their own purposes.

11.1.10 NoAppLog Member

BOOL NoAppLog

Set this member to TRUE to prevent the kernel from writing log messages to a permanent log destination. Messages will still be echoed to the console. Set the value to FALSE for normal logging behaviour.

11.1.11 CreateRepository Member

BOOL CreateRepository

Set this member to TRUE if the kernel should create the repository database as a blank database if it does not already exist, logging will then be directed to this database. Set the value to FALSE then run time initialisation will fail if the repository database is to be used but it does not exist.

11.1.12 RunningAsAddin Member

BOOL RunningAsAddin

Set this member to TRUE if the application is to run as an Add-In Task on a Domino Server. Set the value to FALSE if the application will not run as an Add-In. Applications that are marked as not running as an Add-In can still be executed on a domino server.

11.1.13 NeedsMQ Member

BOOL NeedsMQ

Set this value to TRUE if the application expects to use a Message Queue (MQ) for commands, set it to FALSE if no Message Queue is required.

11.1.14 AllowMultipleAddins Member

```
BOOL   AllowMultipleAddins
```

Set this value to TRUE if multiple copies of this Server Add-In Task are allowed to be run on the same server at the same time. This setting instructs the kernel to create unique Message Queue names for each instance of the Add-In. The Message Queue names are constructed from the application name and a number is appended starting from 1 that is the lowest queue name available on the server. Set the value to FALSE if only a single instance is permitted, in this case the Message Queue name is just the application name and if the name is already in use then kernel initialisation will fail.

11.1.15 LogLevel Member

```
int            LogLevel
```

Set this member to the logging level required for this run of the application. The following symbolic values can be set for the logging level.

- LOGLEVEL_NORMAL – This is the default logging level. At this level errors and exceptions will be written to the log but comparatively few informational messages will be written.
- LOGLEVEL_VERBOSE – At this logging level many more informational messages are written to the application log.
- LOGLEVEL_TRACE – At this logging level additional diagnostic trace messages are written to the application log.
- LOGLEVEL_DEBUG – At this logging level all informational and trace messages are written in addition debugging dumps of certain data object will be written to the application log.

11.1.16 TraceArea Member

```
int            TraceArea
```

If the logging level has been set to TRACE or DEBUG level then this member can be set to restrict the functional areas of the kernel that will write additional messages to the application log. The additional messages from the kernel at these higher logging levels can be very numerous so it is useful to be able to target a specific area of the kernel for debugging. If this value is set to zero then all areas of the kernel will generate the additional messages. The following symbolic values can be used to target specific areas of kernel functionality.

- TRACE_ALL – to trace all areas of the kernel.
- TRACE_CORERT – to trace the core areas of the run time.
- TRACE_MT – to trace kernel areas that specifically support multi-threading.

- TRACE_MTD – to trace the kernel thread dispatcher functional area.
- TRACE_MRQ – to trace the interface between the application and the multi-threading kernel.
- TRACE_EXPLORER – to trace the Domino eXplorer.
- TRACE_CMLS – to trace the Constrained Multi Lane Scheduling functions.
- TRACE_TXH – to trace the functioning of the Transaction Handler.
- TRACE_RESOURCELOADER – to trace the functioning of the Resource Loader.
- TRACE_ACLRSPARSER – to trace the parsing of ACL Rule Sets.
- TRACE_ACLRULESET – to trace the operation of ACL Rule Sets.
- TRACE_DBC – to trace functioning of the Database Copier.
- TRACE_DM – to trace the functioning of the Design Manager
- TRACE_DBM – to trace the functioning of the Database Mover.
- TRACE_DBSCAN – to trace the functioning of the Database Note Scanner.
- TRACE_DBPMOD – to trace the operation of the Database Property Modifier.
- TRACE_NER – to trace the operation of the Named Entity Resolver.
- TRACE_APP – This value is reserved for applications to implement their own additional diagnostic tracing.

11.1.17 szRepServer Member

```
char   szRepServer[MAX_SERVER + 1]
```

If a repository database will be used then set this member to a null terminated string containing the abbreviated name of the server on which the repository database will be found. Set that value to an empty string or the value “Local” if the repository database is on the same server or workstation where the application is executing.

11.1.18 szRepDb Member

```
char   szRepDb[MAX_DATABASE + 1]
```

If a repository database will be used then set this member to a null terminated character string containing the path of the repository database relative to the Notes Data Directory.

11.1.19 APPName Member

```
char   APPName[MAXAPPNAME + 1]
```

Set this member to the name of the application.

11.1.20 APPTitle Member

```
char   APPTitle[MAXAPPTITLE + 1]
```

Set this member to a short title for the application.

11.1.21 APPVer Member

```
char   APPVer[MAXAPPVERSION + 1]
```

Set this member to the version of the application, it is suggested to differentiate between different build configurations of the application.

11.2 ThreadManagerPolicy Class

An object of this class contains information used by the thread manager to configure the multi-threaded runtime system. An application can configure an object of this class and use it in the creation of the runtime system to influence many settings and constraints that are used by the runtime system. Only members that should be set by the application are described here, other members of the class are intended for internal use by the kernel. For a more detailed description of the use of the members of this class refer to the section on the “Threading Model” later in this document.

Header File: DXCommon/Threads/ThreadManagerPolicy.h

11.2.1 Object Constructor

```
ThreadManagerPolicy(void)
```

11.2.1.1 Returns

A pointer to the newly created ThreadManagerPolicy object.

11.2.2 TPSchedMode Member

```
UINT TPSchedMode
```

This member determines the request scheduling mode that is to be employed by the kernel. Set the member to one of the following symbolic values according to the application model.

- TPOOL_MODE_USLS – This sets the scheduling mode to Unconstrained Single Lane Scheduling which treats the Thread Pool as a single resource where any request can be executed by any thread.
- TPOOL_MODE_CMLS – This sets the scheduling mode to Constrained Multi Lane Scheduling. This mode reserves a number of members of the Thread Pool for executing particular types of request.

11.2.3 TPoolPolicy Member

```
UINT TPoolPolicy
```

This member enables policy options for the thread scheduler. Set this member to the symbolic value `TPOOL_POLICY_PRESTARTTARGET.`

11.2.4 PriorityPolicy Member

```
UINT PriorityPolicy
```

This member determines policy options for managing the relative prioritisation of requests by the Thread Scheduler. Use the following symbolic values to determine the options in effect.

- PRIO_POLICY_AGERQS – If this setting is in effect then requests that are waiting to be executed will be examined periodically and their priority will be incremented each time.
- PRIO_POLICY_PREFBOOST – If this (and the above) setting are in effect then requests that are waiting to be executed will be examined periodically and their priority will be increased (boosted) by a value determined by a policy setting.

11.2.5 TargetThreads Member

```
int TargetThreads
```

This member determines the number of threads that will be used in the Thread Pool. Set the MaxThreads and MinThreads members to the same value. A minimum value of 10 threads is suggested.

11.2.6 PendingRQECapacity Member

```
int PendingRQECapacity
```

This member determines the size (number of requests) of the “Pending Requests” Pool, this pool stores requests that are waiting to be executed. A suggested value for this is 20 * Number of Threads.

11.2.7 RejoinRQECapacity Member

```
int RejoinRQECapacity
```

This member determines the size (number of requests) of the “Rejoin Requests” Pool, this pool stores completed requests that are waiting to be polled by the application. A suggested value for this is 20 * Number of Threads.

11.2.8 AsyLogPoolEntries Member

```
int AsyLogPoolEntries
```

This member determines the size (number of log messages) that can be stored waiting to be written. The appropriate setting for this member will vary according to how much logging is generated by the application and the logging level that is in effect. A suggested value for this setting is 200.

11.2.9 MaxPctL0Threads Member

```
int MaxPctL0Threads
```

If the scheduling mode is set to “Constrained Multi Lane Scheduling” then this member determines the maximum percentage of threads in the thread pool that can be used by Lane 0 (service requests).

11.2.10 MaxPctL1Threads Member

```
int MaxPctL1Threads
```

If the scheduling mode is set to “Constrained Multi Lane Scheduling” then this member determines the maximum percentage of threads in the thread pool that can be used by Lane 1 (feeder requests).

11.2.11 MaxPctL2Threads Member

```
int MaxPctL2Threads
```

If the scheduling mode is set to “Constrained Multi Lane Scheduling” then this member determines the maximum percentage of threads in the thread pool that can be used by Lane 2 (unit transaction requests).

It is suggested that sum of L0, L1 and L2 threads in the pool does not exceed 50%.

11.3 TransactionQueue Class

This class provides information that is used by the TransactionHandler to bind to the physical implementation of a transaction queue and determine a number of operational characteristics of the queue.

Header File: DXCommon/MTX/TransactionQueue.h

11.3.1 Object Constructor

```
TransactionQueue(void)
```

11.3.1.1 Returns

A pointer to the newly created TransactionQueue object.

11.3.2 wQueueProtocols Member

```
WORD   wQueueProtocols
```

This member determines how the transaction handle will manage this queue. The value consists of a number of bit flags, use the following symbolic values to set the required settings.

- `QPFLAG_REQUEIP_STARTUP` – If this flag is set then when the transaction queue is started then any transactions that are on the “In Progress” queue will be requeued for execution.
- `QPFLAG_REQUEIP_READY` – If this flag is set then any “In Progress” transactions that are requeued during startup will be moved to the “Ready” queue for immediate execution. If the flag is not set then requeued transactions will be moved to the “Delayed” queue for later execution.
- `QPFLAG_REQUE_DELAY` – If this flag is set then the “Delayed” queue will be monitored for transactions and they will be requeued, how they are requeued is determined by the following two flag settings.
- `QPFLAG_DELAY_IDLE` – If this flag is set then whenever the “Ready” queue is empty the “Delayed” queue will be checked for any transactions that can be requeued.
- `QPFLAG_DELAY_CYCLE` – If this flag is set then after every n transactions are processed a check will be made of the “Delayed” queue to detect transactions that can now be requeued.
- `QPFLAG_MONITOR_SCHED` – If this flag is set then the “Schedule” queue will be checked for any transactions that should be executed on a timed basis.

To set the values for default processing configuration use the `QPFLAG_DEFAULT` value that will set the `QPFLAG_REQUEIP_STARTUP, QPFLAG_REQUE_DELAY and QPFLAG_DELAY_IDLE` flags.

11.3.3 MaxConcurrent Member

```
int    MaxConcurrent
```

This member determines how many transactions are allowed to be executing concurrently.

11.3.4 MaxRunLimit Member

```
DWORD  MaxRunLimit
```

This member determines the maximum number of transactions that can be run from this queue, once the limit is reached the queue will automatically shut down.

11.3.5 hdbQueue Member

```
DBHANDLE      hdbQueue
```

This member can optionally be set to the database handle for the queue database, if the handle is not set then the transaction handler will use the server name and database path to open the queue database.

11.3.6 MinDelay Member

```
int    MinDelay
```

This member holds the minimum length of time in seconds that a transaction must remain on the “Delayed” queue before it is eligible to be re-queued.

11.3.7 ReQTXCycle Member

```
int    ReQTXCycle
```

This member contains the number of transaction to process before checking the “Delayed” queue.

11.3.8 MaxReqCount Member

```
int    MaxReqCount
```

This member contains the maximum number of “Delayed” transactions that will be re-queued in any cycle.

11.3.9 DelayCycleSecs Member

```
int    DelayCycleSecs
```

This member contains the number of seconds between regular inspections of the “Delayed” queue.

11.3.10 LocalPermit Member

```
BOOL   LocalPermit
```

This member is the permit that allows the transaction handler to continue processing this queue. The application should set this value to TRUE before starting to process this queue and should set the value to FALSE when the transaction handler should shut down processing this queue.

11.3.11 QueueIsSuspended Member

```
BOOL   QueueIsSuspended
```

Set this value to TRUE to temporarily suspend processing of transactions from this queue. Set the value back to FALSE when transaction processing can resume on this queue.

11.3.12 szQueueName Member

```
char   szQName[MAX_ELEMENT + 1]
```

This member should be set to a null terminated character string providing the logical name of this transaction queue.

11.3.13 szQServer Member

```
char   szQServer[MAX_SERVER + 1]
```

If a database handle is not supplied for the transaction queue, this member should be set to a null terminated character string containing the abbreviated name of the server on which the transaction queue database resides. Set the value to an empty string or the value “Local” if the transaction queue database is on the same server or workstation where the application is running.

11.3.14 szQDbPath Member

```
char   szQDbPath[MAX_DATABASE + 1]
```

If a database handle is not supplied for the transaction queue, this member should be set to a null terminated character string containing the path of the transaction queue database relative to the Notes Data Directory.

11.3.15 szReadyQName Member

```
char   szReadyQName[MAX_ELEMENT + 1]
```

Set this member to a null terminated character string containing the view name of the “Ready” queue in the transaction database. This member has a default value of “NewTransactions”.

11.3.16 szInProgressQName Member

```
char   szInProgressQName[MAX_ELEMENT + 1]
```

Set this member to a null terminated character string containing the view name of the “In Progress” queue in the transaction database. This member has a default value of “InProgressTransactions”.

11.3.17 szDelayedQName Member

```
char   szDelayedQName[MAX_ELEMENT + 1]
```

Set this member to a null terminated character string containing the view name of the “Delayed” queue in the transaction database. This member has a default value of “DelayedTransactions”.

11.3.18 szSchedQName Member

```
char  szSchedQName[MAX_ELEMENT + 1]
```

Set this member to a null terminated character string containing the view name of the “Scheduled” queue in the transaction database. This member has a default value of “ScheduledTransactions”.

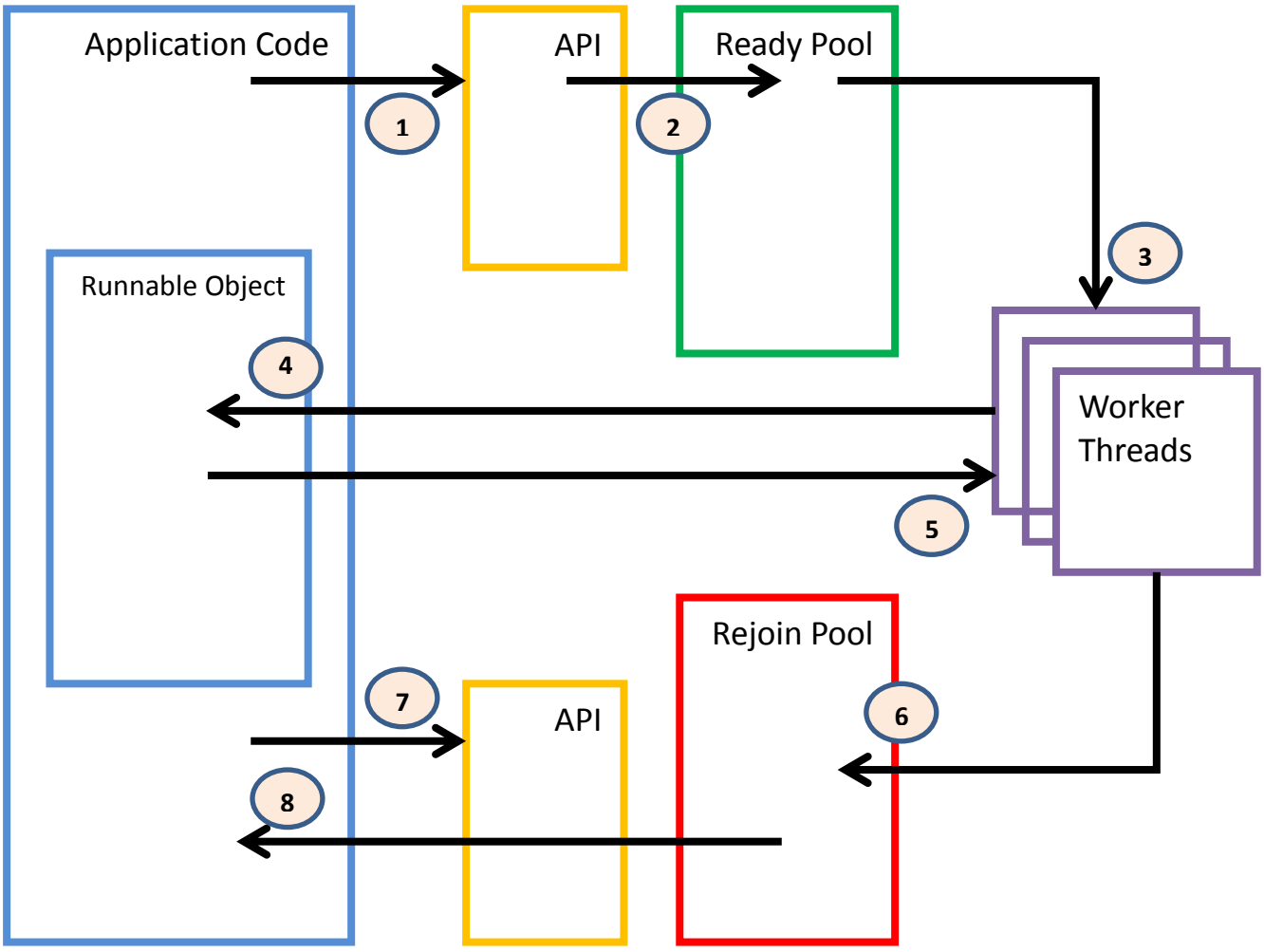
12. Threading Model

This section of the document presents the key aspects of the threading model that a developer using the API should be familiar with.

12.1 Introduction

The DX threading model has been designed to present application developers with a simple architecture that is easy to design for and an API that is simple to use. The application interface is based on a message passing interface (MPI). Applications create request objects that describe some work that must be performed asynchronously and post these request to the kernel for execution. The kernel manages a pool of threads, the threads in the pool are homogeneous and can execute any request. The kernel will dispatch requests for execution by one of the worker threads in the thread pool. Once a request has completed the state change can be detected in the application code by a polling mechanism that is invoked through the API.

12.2 The Request Lifecycle



The application creates a request object and populates with the information needed to execute a chunk of work. The application the calls the “PostARequest” function in the run time API, passing the address of the request object and the address of the “Runnable” object that is to execute the request.

The run time API takes the information passed by the application code in the “PostARequest” call and stores it in the “Ready Pool” where it is available to be executed.

The kernel code monitors the pool of worker threads and as soon as one is available to run work it will locate the most appropriate request that is waiting in the “Ready Pool” and will dispatch it to the available worker thread for execution.

The worker thread will invoke the “ExecuteThisRequest” interface on the Runnable object to have the application code service the request. The application code will indicate the success or otherwise and return any needed information in the request object that was passed to it.

When processing of the request is completed the application code returns to the Worker Thread.

The Worker Thread stores the information in the “Rejoin Pool” and signals the kernel that it is available for processing work again.

The application code calls the “GetRejoinRequest” function in the run time API to poll the “Rejoin Pool” to see if request have completed processing.

If a request has been completed then the API will return the address of the completed request object. The application code then processes any returned information and disposes of the returned request object.

12.3 The Request Owner

Calls to the “PostARequest” and “GetRejoinRequest” functions in the run time API take a parameter of “Request Owner” this parameter is an arbitrary address encoded as a void pointer (void *). This parameter provides a mechanism for grouping bunches of requests together and localising the code that will process these grouped requests.

The kind of processing that the DX kernel was designed for often breaks down into a hierarchic pattern for parallel execution, one request will generate a number of sub-requests and each sub-request will, in turn, create a number of sub-sub-requests and so on. The owner mechanism can be used here to reflect the hierarchic workload, in this case the Owner for each request is set to the address of the parent request in the hierarchy. When polling for completed requests using the “GetRejoinRequest” the address of a parent request is specified as the owner and the return will signal when every sub-request that belongs to that parent has completed and therefore the parent processing can be completed.

12.4 Request Priority

Calls to the “PostARequest” functions in the run time accept a parameter that specifies the “Priority” of the request. The priority is specified as an arbitrary integer value with larger numbers being a higher (more urgent) priority. The priority is used by the kernel to determine which of the requests available in the “Ready Pool” will be the next to be dispatched to an available thread.

As a general rule workloads that follow the hierarchic model described in the section above should post requests at higher priorities the lower they are in the hierarchy.

The kernel also implements an optional, request priority ageing mechanism. When ageing is in effect then requests that reside in the “Ready Pool” have their priority increased at regular intervals. This mechanism is intended to prevent requests becoming stale while waiting to be executed and tying up resources while not contributing to throughput rates.

12.5 Constrained Multi Lane Scheduling

As pointed out in the earlier sections the kernel treats all worker threads as equals, any request can be dispatched to any worker thread that is available to process work. When a single request is being processed by a worker thread all processing for that request must be completed before the thread becomes available to process other requests, including the execution and rejoin processing of any sub-requests that are posted. There is a fundamental exposure from this model, it is possible for all threads to fill up with “higher level” requests leaving no worker threads available to execute the lower level requests that have been posted. In this scenario processing will simply grind to a halt with all worker threads waiting for sub-requests to complete, which they never will, or waiting for space to become available in the “Ready Pool” so that more sub-requests can be posted.

The kernel solves this thread exhaustion problem by providing a different scheduling mode “Constrained Multi Lane Scheduling” (CMLS). The CMLS mode is selected by setting the TPOOL_MODE_CMLS bit in the TPSchedMode member of the ThreadMnagerPolicy object that is used to configure the multi-threading kernel.

When running in CMLS mode the kernel still regards all worker threads as being equal and able to execute any request however it limits (constrains) the number of threads that can be concurrently executing requests from different levels in the workload hierarchy. CMLS identifies four arbitrary levels of request hierarchy. Level or Lane 0 defines service requests these requests would normally be running for the duration of the application. Level or Lane 1 defines requests that will themselves generate any number of what the application would recognise as unit transactions, these are referred to as feeder transactions. Level or Lane 2 defines unit transactions and Level or Lane 3 defines sub-requests or requests that will perform the work of a part of a transaction. The Level or Lane for an individual request is identified to the kernel by setting the appropriate bits in the Attributes flag that is passed in the call to “PostARequest”.

Constraints may be applied as a percentage of the threads in the thread pool that can be executing requests from levels 0, 1 and 2. These constraints are applied by setting the appropriate members in the ThreadManagerPolicy that used to initialise the kernel. The constraints not only apply to the worker threads but also to the number of requests in that level that can be in the “Ready Pool” at any point in time. The protocols for the CMLS implementation allow a worker thread or a “Ready Pool” entry to be used from the requests level or from a resource that is available from any higher level.

Supposing that there is an application that will execute with 10 worker threads in the thread pool and 100 entries in the “Ready Pool”, the application has configured the CMLS limits as Lane 0 is set to 0% (i.e. we will not be executing any of these requests, Lane 1 is set to 20% and Lane 2 is also set to 20%. In this example there could be a maximum of 20 Lane 1 requests in the ready queue at any point in time and there could be a maximum of 2 Lane 1 requests executing concurrently. There could also be a maximum of 40 Lane 2 requests in the “Ready Pool” at any point in time, assuming that there were no Lane 1 requests in the pool at that time and there could be a maximum of 4 Lane 2 requests executing concurrently, also assuming that no Lane 1 requests were executing at that time. Lane 3 requests are always unconstrained and can occupy all of the available slots in the “Ready Pool” and can be concurrently executing requests in every thread in the thread pool.

It should be noted that the priority mechanisms described in the previous section remain in effect when the CMLS scheduling mode is engaged.

The original analogy used in the design of the CMLS facility was to view the worker threads as separate lanes on a motorway and to view the different Levels as vehicle types with 0 being large articulated lorries, 1 being lorries, 2 being vans and 3 being cars and motorbikes. Signals above the motorway restrict vehicle types to only using assigned lanes. When entering the motorway, if the assigned lanes for your vehicle type are full then you have to wait. The analogy can still be useful but does introduce some false assumptions about how CMLS works. The main failing is that threads are not assigned to handle particular CMLS levels, individual threads can be used for any request but CMLS will prevent the total current work profile from exceeding any of the prescribed constraints.

12.6 The Design of Runnable Classes

Although there are no real constraints imposed by the kernel for the design of Runnable objects apart from the fact that they need to inherit from the “Runnable” class and implement the “ExecuteThisRequest” interface, there are a few simple guidelines that should be followed to ensure a successful implementation pattern.

There should never be a need to instantiate more than a single instance of any runnable class, no matter how many threads are being run in the thread pool. Some developers assume that there is some kind of affinity between the Runnable object and a particular thread in the pool, this is not the case there is no such affinity.

Any variable data used in processing a request should only be held in either local automatic storage or in “Transaction Storage” i.e. members in the request object. These variables should **NEVER** be stored in members in the Runnable object. The selection of Automatic or Transaction is determined by the lifecycle of the data in the variable. If the data is to be used across multiple asynchronous request dispatches then the variable should be stored in Transaction storage, if the data is only to be used for the processing of a single request then it is probably more appropriate to use Automatic storage.

The kernel does not have any provision for “Thread Local Storage” i.e. memory that is reserved for use by a single thread. The kernel does provide API functions for using one resource, database handles, on a per thread basis. All other Domino resources can be used from multiple threads, compiled formulas can only be used by one thread at a time but are more appropriately handled in Transaction storage rather than dedicated to a particular thread or by serialising access to the compiled formula.

Only design implementation aspects that are directly pertinent to using the API are presented here, for more information on designing and building applications using DX refer to the publication “DX Tools Application Design Guide”.

12.7 Request Sizing

The DX threading model has been designed to handle large workload tasks with heavy I/O requirements (network and disk), high memory occupancy and moderate CPU processing, we use the term “Heavy Lift Computing” (HLC) for these types of workload. The model is absolutely **NOT** suitable for the implementation of “High Performance Computing” (HPC) applications.

To ensure that applications fit the “Heavy Lift” paradigm it is important to design the lowest level sub-requests used in the application so that they do not contain too small a quantum of the total workload. There are no definitive rules to determine what is the optimal size and characteristics of the lowest level sub-requests, determining this is a part of the application tuning process. The most successful approach has been to identity the smallest sensible unit of processing at the lowest level of functional decomposition and then to make the lowest level sub-request capable of processing a variable number of these base functional quanta. Tuning of the application consist of changing the number of threads in the pool and varying the number of base functional quanta in the lowest level sub-requests, alongside eliminating bottlenecks and resource contention.

The “Database Copier” (DbCopier) engine implements a good method for dealing with the sizing of the lowest level requests. The functional quantum in the copier is a request to copy a single note from the source database to the target database, the engine determines a value for how many quanta will be combined into a sub-request by computation using size of the source database and the number of documents to be copied. Databases that have many small documents will dispatch sub-requests with more document copy operations than when copying databases with fewer larger documents. The Database Copier also implements a mechanism for scaling the copy operations per request count by a specified factor, this allows for rapid tuning of an implementation.

It has also been noted that if the functional quantum in an application has long wait times associated with it, such as disk I/O to very slow devices or more usually network I/O over “long fat pipes” then these benefit, from running more threads with a smaller size of sub-request.