



Telephony Application Architectures for Intel NetStructure® Boards with DM3 Architecture

Table of Contents

Executive Summary	1
DM3 Architecture Overview	1
Suitable Application Architectures	2
Advantages of Single Process Design	2
Programming Models	2
Synchronous	3
Asynchronous and Its Variations	3
Multithreaded Architectures	4
Single Process	4
Multiprocess	6
Nuance Audio Provider and the DM3 Architecture	7
What Is the Nuance Audio Provider?	7
Using the Nuance Audio Provider	8
Sharing CSP Voice Resources	9
Dynamic vs. Static Device Allocation	9
Moving from Springware to DM3 Architecture	10
Identifying Multiprocess Problems	10
An Annotated Bibliography of DM3 Reference Information	11
Application Notes	11
Technical Notes	11
Intel Documentation	12
Nuance Documentation	13

Executive Summary

When porting an existing application from Springware to DM3 architecture or designing a new application for DM3 architecture, guidelines are helpful, especially about the correct use of processes and threading and the handling of individual channels.

This application note will not fully cover all aspects of the use and programming of applications using DM3 architecture building blocks, but will focus on how an application should be structured to best take advantage of the DM3 architecture and avoid pitfalls that might lead to wasted development time or problems with an application after it is deployed. It also provides an annotated bibliography that helps locate information related to DM3 architecture in other application notes and in documentation.

The information in this application note can be applied to telephony applications running on either Windows* or Linux* systems.

DM3 Architecture Overview

DM3 architecture consists of embedded software modules and Intel® Dialogic® and Intel NetStructure® hardware that provide a platform for developing call processing applications. The block diagram in Figure 1 shows a high-level view of the components in the host-based software stack of the DM3 architecture. These modules range from the operating system drivers up to the telephony application itself.

Application programmers can use Figure 1 and the definitions of the layers given below for reference. However, they cannot access these modules directly. An application can only access media resources and call control functions provided by the DM3 architecture through the R4 API.

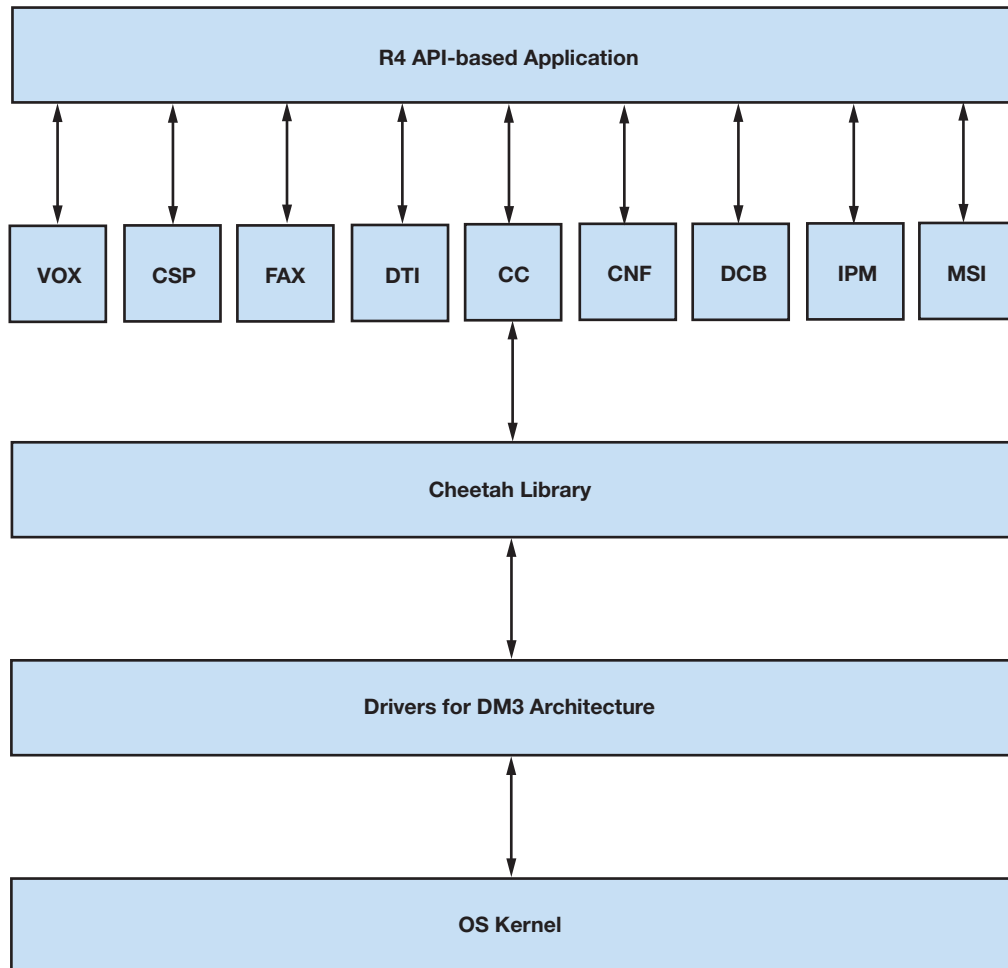


Figure 1. Block Diagram of Software Stack for DM3 Architecture

Layers in the software stack contain the following:

- **R4 API-based Application** — User-written telephony application containing R4 API calls to Intel hardware using the DM3 architecture and the Standard Runtime Library (SRL)
- **VOX, CSP, FAX, DTI, CC, CNF, DCB, IPM, and MSI** — R4 API module groupings that are the highest level of translation between the R4 API and the DM3 architecture
- **Cheetah Library** — Layer containing translations from the R4 API to the native low-level messages used when implementing an application on hardware using the DM3 architecture with Windows or Linux
- **Drivers for DM3 Architecture** — Operating system-level device driver that controls hardware with DM3 architecture using PCI or CompactPCI*
- **OS Kernel** — Windows or Linux kernel-level constructs that underlie the device drivers for the DM3 architecture

Suitable Application Architectures

Successful application design architectures using the DM3 architecture are more restrictive than design architectures that work well with Springware architecture. Older, non-multithreaded operating systems, prevalent when the Springware architecture was first designed, drove application design architecture toward a single-process-per-channel model. Coordination between processes was provided by an application-defined inter-process communication (IPC) mechanism.

Springware architecture could also be used to allow more than one process to access the same channel on a board. With the DM3 architecture, multiprocess device sharing is generally not allowed or recommended. (See Section 6.4, “Additional DM3 Considerations — Multithreading and Multiprocessing” in the *Voice API Programming Guide* for more information about this subject.) However, some situations require an exception, such as designing a successful interface with Automatic Speech Recognition (ASR) and Text-to-Speech (TTS) servers. A speech application using continuous speech processing (CSP) is an interesting example.

Speech architectures using CSP allow the sharing of a voice channel between processes (or applications) when one of the following is used: a board with Springware or DM3 architecture or Intel NetStructure® Host Media Processing Software Release 1.3 or higher. The following rules apply:

- Playing or recording/streaming from both processes is not supported. One process must do the play function and the other process must do the record/stream.
- Both processes must coordinate operations on shared devices via some method of inter-process communication. These operations include opening/closing, setting parameters, timeslot routing, and event processing.
- Both processes cannot open and close the same device simultaneously. These functions must be done sequentially.
- Both processes must ensure that parameters set on the same channel (using setparm) go to the same component on the board and overwrite each other in the correct order. This rule also applies to timeslot routing. The last process that sets a parameter or routes a timeslot will take effect by overwriting the operation done by the previous process.
- Events are delivered to the specific process that starts the corresponding media function. To share events among processes, inter-process communication is required.
- Voice channel sharing only works with CSP. One process must support CSP only while the other process performs media play. No other scenario is supported.

Advantages of Single Process Design

While only allowing a single process to access a channel may at first seem restrictive, it actually allows a more efficient application design, which is necessary to handle the higher channel densities possible in a system using DM3 architecture. Because newer multithreaded operating systems and the DM3 architecture itself call for a different approach, single-process, multithreaded asynchronous design can lead to applications that are easier to build and maintain and result in a lower cost of ownership. Some advantages are:

- Asynchronous design leads to the use of state machines for application and device logic. An application based on such a design is much more flexible and easier to maintain in the long run than a design implemented using the kind of nested switch statements that are often found in synchronous program design.
- Much of the inter-process communication needed to move data between processes and coordinate uncoupled logic is eliminated in a multithreaded application.

- Relying on an operating system's optimized thread scheduling is more efficient than relying on process scheduling.

Programming Models

A brief survey of synchronous and asynchronous programming models in relation to the DM3 architecture will be provided here. For more information on this subject, see the sources listed in the annotated bibliography at the end of this application note. Of special interest is the in-depth discussion of programming models and their performance characteristics with DM3 architecture in *Intel® Dialogic® System Release 6.0 on PCI for Windows* Standard Runtime Library – Optimizing Performance*, which can be found at <http://resource.intel.com/telecom/support/appnotes/9029/index.htm>

Synchronous

Calling Intel APIs in synchronous mode blocks process or thread execution until the API function completes. For a lengthy operation such as playing an audio file, blocking could occur for many seconds, preventing the process or thread from doing any useful work during that time. Because of this delay, a program design that handles multiple simultaneous channels becomes impossible, and each device would require a separate application, process, or thread. As individual processes and threads “sleep,” the operating system would continue to process unblocked ones. When the synchronous function completes, the process or thread “wakes up” and processing continues.

While a synchronous approach neatly segments each channel into a single process or thread, it has the disadvantage of being very inefficient. A synchronous model consumes significant system resources for process/thread scheduling and context switching, making it unsuitable for a high-density system with DM3 architecture. For example, E-1 configurations running R2MF have been observed to experience performance problems when running a single process per channel for 60 channels. Such problems would prevent an application from reaching a common application density goal of eight T-1 spans (192 channels).

When creating applications with DM3 architecture, it is best to avoid the synchronous programming model except for low-density demonstration programs or very simple applications that are not intended for deployment.

Asynchronous

When Intel API calls are performed in asynchronous mode, the process or thread continues while the API function completes. Only a short delay of milliseconds takes place when the API call returns and the process or thread can continue executing. When the API call operation ends, the application receives the completion event asynchronously.

Asynchronous operation allows multiple channels to be handled easily in a single application, process, or thread, and uses system resources more efficiently. It also requires an event-driven application and the use of a state machine. Advantages are the reduction of system overhead required for inter-process communication and thread scheduling and the ability to manage multiple devices easily, which results in the possibility of higher channel densities.

In addition, the DM3 architecture is asynchronous by design, making asynchronous application architecture a better “fit” to the way in which the boards and their software function.

Asynchronous Polled and Callback Models

Asynchronous polled and callback models seem to differ at first glance. In the polled model, the application waits for the event using an event-loop to retrieve and dispose of events. With the callback model, event-handling callback functions are enabled for specific events on specific devices. The event handler(s) are then automatically called when an event arrives from a device for which an event handler is specified.

In reality, both polled and callback asynchronous models work the same way because in the callback model, the event loop is maintained within the SRL rather than by the application. The polled model may allow a little more flexibility in specifying polling timeouts, but the callback model is easier to use.

Extended Asynchronous Model

The difference between the extended asynchronous model and the basic asynchronous model is that different groups of devices may be assigned to different event handlers in the extended model. When using the extended model, multiple threads can be created, each of which controls multiple devices. In such an application, each thread may have its own state machine for the type of device that it controls. For example, one group of devices may provide fax services and another group speech (ASR and TTS) services, while both share the same process space and database resources.

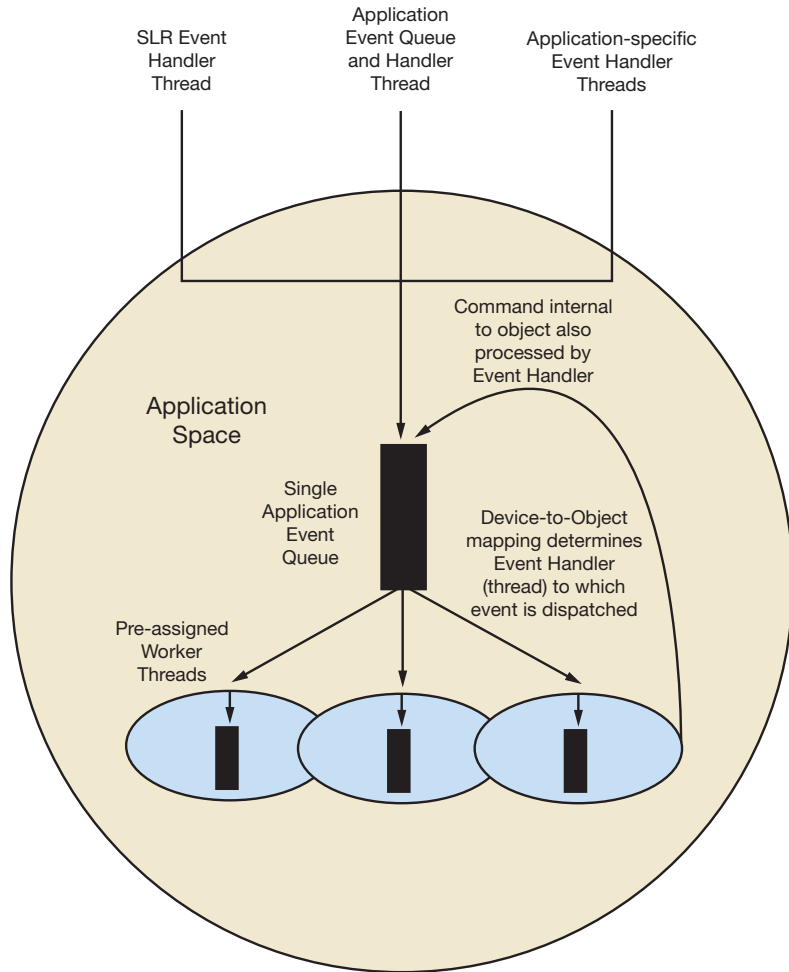


Figure 2. Single-Process, Multithreaded Architecture

Multithreaded Architectures

To achieve the highest channel density possible with DM3 architecture, single-process, multithreaded application architecture is recommended. Such a design may vary slightly in specific applications, but the sample architecture described in the “single-process” section below should be applicable to most telephony applications.

Single Process

In a multithreaded architecture using a single process, the application’s threads are organized around event queues and the handling of events in these queues as shown in Figure 2.

Four types of threads are used in this architecture.

SRL Event Handler Thread

The SRL Event Handler Thread has a single function — fetching asynchronously generated events from an internal SRL event queue and placing them on a centralized application event queue. All event data must be copied

out of the SRL structures into suitable application containers before retrieving the next SRL event. If this is not done, data elements may be overwritten.

A number of event interpretation-related API calls are used to fetch additional event data. These calls include `gc_GetUsrAttr()`, `gc_GetSigInfo()`, `gc_GetUsrInfo()`, and `gc_ResultInfo()`. These call routines must also be called in the same thread before retrieving the next event from the queue. If this recommendation is not followed, no problems may occur in a lightly loaded system. However, timing in a heavily loaded system can be very different, and difficult-to-reproduce bugs involving corrupted data may occur.

Events originating from the SRL also need further mapping so that the network or media object they relate to can be identified. A Registration Manager may be used for this function because it maintains a list of SRL device handles mapped to the objects that handle the processing for each channel or device.

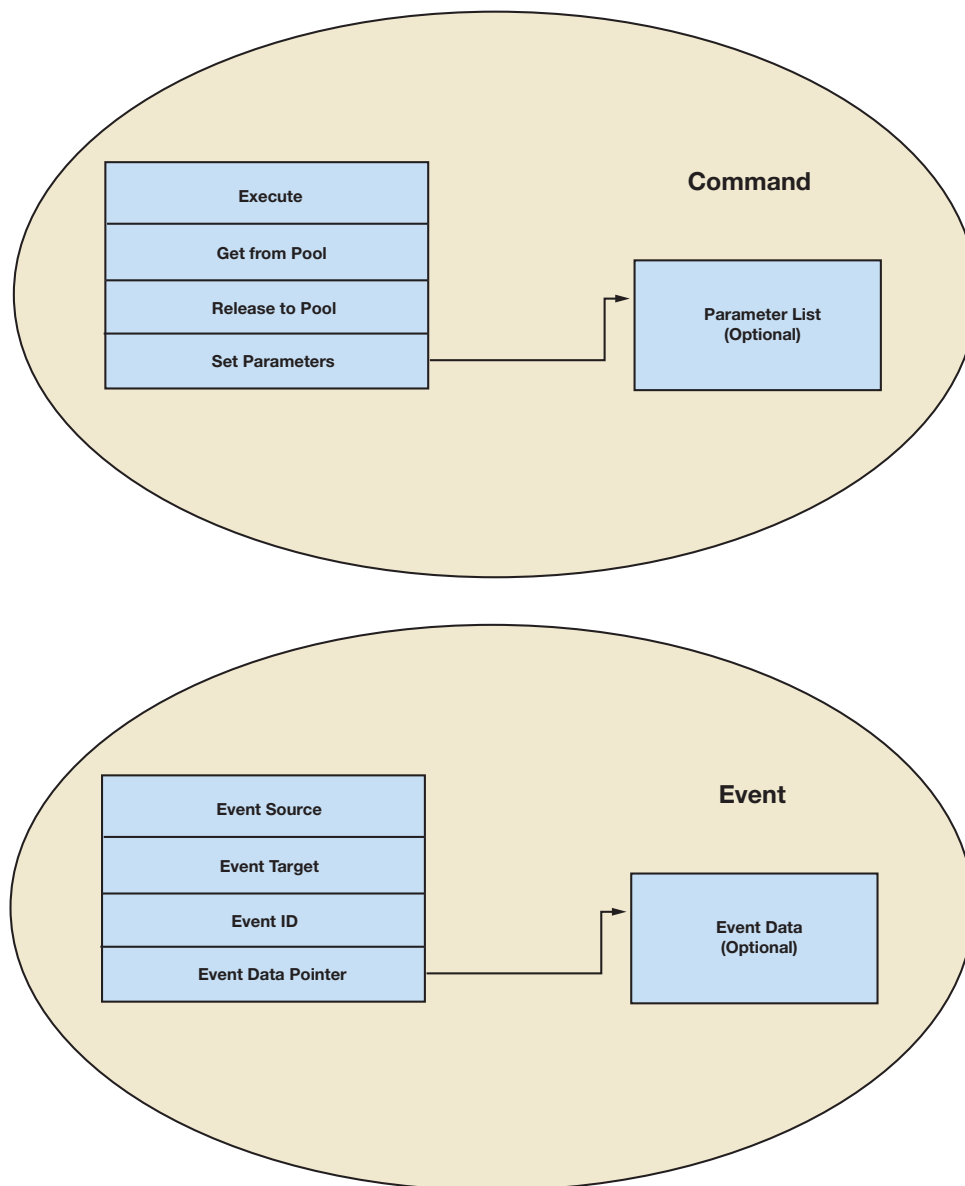


Figure 3. Command and Event Objects

Application-specific Event Queue and Handler Threads

The Application-specific Event Queue and Handler Thread implements an event queue that holds SRL and application-oriented events. Events can include asynchronous notification of data fetched from an application's database, recognition events from an ASR server, or events that allow a web-based GUI to control an application. The main state machine, which contains the application logic, works with this event stream.

Pre-assigned Worker Threads

Pre-assigned worker threads use an event model that is an implementation of the C++ Command pattern. The model handles both commands and events while also controlling data synchronization.

The Command pattern usually consists of a client, which is the source of commands, a receiver, which is the target of commands, and the invoker, which is the mechanism by which commands are executed. Command objects replace direct function calls in this architecture.

The Command pattern may also be extended to include events. These events are specialized commands in which the Execute() function of the event class performs an action appropriate for the event type or simply passes the event data to its destination as a parameter in an event handling function. Commands and events are considered symmetric and are modeled as illustrated in Figure 3.

This event model addresses event serialization and data synchronization by using worker threads whose main function is to process events. Associated with each event processor thread is a queue that holds event and command objects. Each target object managed by the application (a network or media device, for example) is assigned an event processor by the application at instantiation. This allows the application to determine how many threads to use and which set of objects are handled by each thread.

Event and command objects are queued to the event processor and processed in the order received. The queue is thread-safe so that events and commands may be queued from any thread. The client source obtains and initializes a command or event object and queues it to an event processor. Since no command and event processing is done in the source's thread, a minimum number of execution cycles are used. This can be critical in situations where the source thread has soft real-time response requirements, such as a thread that communicates with a network driver.

The event processor worker thread waits for a condition variable, which is triggered when items are added to the queue. When an event or command item is added, the worker thread pops the command or event object from the queue and calls the object's `Execute()` function. This provides event serialization. Events and commands are processed in the order they are received. Because all command and event processing for an object is performed in the worker thread of its assigned event processor, data synchronization mechanisms at the data-element level, normally associated with multithreaded applications, are not necessary. Data synchronization for the entire object is provided by the event processor queue.

The application architecture described here can have any number of event processors. Each command target or event handler object is assigned an event processor as part of instantiation. The number of event processors, and which objects use each, are under application control. The optimal number of event processors and their use is dependent on application characteristics and is most effectively determined by experiment.

Stateful handling of commands and events is also critical to a robust telephony application. In an asynchronous environment, objects in all software layers must maintain state to successfully handle both valid and invalid operations, and all commands and events must be handled in a deterministic fashion. An appropriate action

is taken for commands and events valid for a particular state, and all other commands and events are handled as errors.

Object state must be maintained and checked. A media player is a simple example. The media player could be in any of the following states: Opening, Idle, Playing, or Closing. If a command to play a file arrives at the player object when it is in any state other than Idle, the command is not issued to the hardware, and an error is generated. If it is in an Idle state, its state is advanced to Playing, and the hardware is told to play the file. A play termination event would then move the player object from a Playing state back to Idle.

A state machine for a network object would be considerably more complicated, but would work in the same way.

Application-specific Event Handler Threads

A particular application may require other threads. These may include interfaces to external systems or ways to handle application tasks such as event or error logging.

Multiprocess

In some cases, an application designer will hesitate to rely on a single process for all channels. The usual reason is that, if the process crashes, all channels will be unavailable, and the system will be useless until it is restarted.

However, if a single process has been thoroughly load tested and does not contain any memory leaks or other latent problems, a single-process design is no less reliable than multiple instances of this same process handling a smaller number of channels. Subdividing an application only seems "safer."

A multiprocess, multithreaded model pays a performance penalty in system overhead used for operating system context switching between its processes. The size of the penalty depends on the number of processes and threads per process. As a general rule-of-thumb, an application may be divided into multiple processes without serious performance penalties down to no less than a per-span basis, that is, 24 to 30 channels per process.

A multiprocess, multithreaded model is most suitable for fixed routing situations where network and media devices remain joined to one another throughout an application instead of being drawn from pools of available devices. Also, the number of devices available in a pool is finite, and pool sizes are generally smaller when there are a

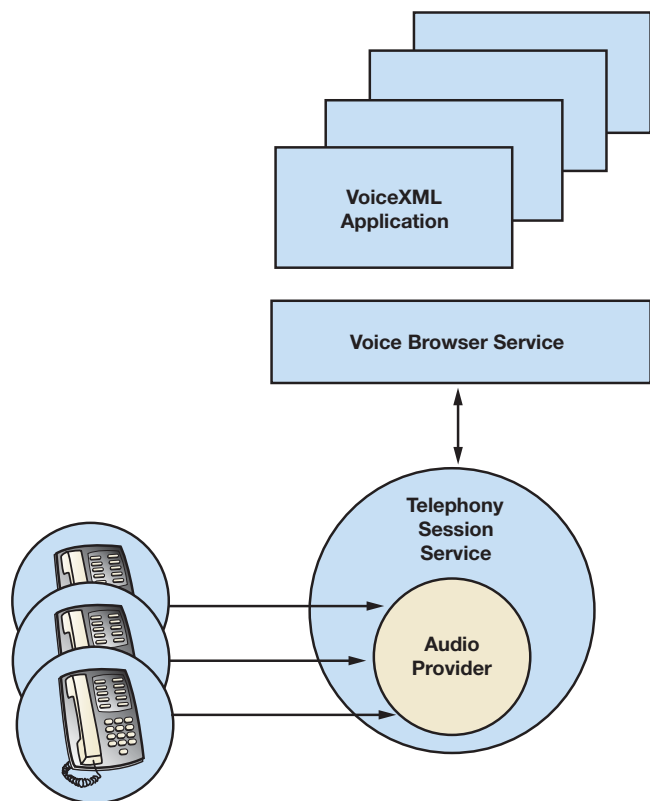


Figure 4. Application Architecture with Nuance Audio Provider

amount of time does pass between the asynchronous close and the time when another process can safely use the same device. Load and timing-dependent bugs may result. For example, a firmware “glare” condition can result in an application that has two concurrent processes using `dx_open` and `dx_close` for device discovery. Each process cycles through a list of likely devices, and checks for a device by opening and closing it. These two processes can sometimes attempt a simultaneous close for the same device. Since DM3 architecture is not designed to respond to two concurrent attempts to control the same device, the `dx_close` that arrives second is ignored by the firmware and the API call that initiated the `dx_close` can take up to 15 seconds to return at the library/application level.

Nuance Audio Provider and the DM3 Architecture

When using the DM3 architecture, a single-process application design is preferable. However, in cases where ASR and TTS are used, such as with the Nuance audio provider, it is necessary to divide device access into more than one process.

What Is the Nuance Audio Provider?

The Nuance audio provider is software that serves as an interface layer between the Nuance system and the R4 APIs from Intel. This interface layer supports telephony control, recording, playback, and DTMF detection for a range of telephony interface products, including products with DM3 architecture.

The Nuance software layer allows considerable configuration flexibility, from full control of all audio provider functionality to the simple recording capability needed for speech recognition. The audio provider may be configured as a complete telephony solution or it may coexist with other Intel call control software in an integrated solution.

Figure 4 presents a typical telephony application architecture that includes the Nuance audio provider.

The application logic in Figure 4 is implemented using the Voice Extensible Markup Language (VoiceXML or VXML), which resides on a system separate from the server that is responsible for call processing. The call processing component consists of a custom-written telephony application, which uses an Intel call control API such as Global Call. This telephony application, in turn, uses of the Nuance audio provider as its interface to speech recognition and other media capabilities.

larger number of pools. A situation in which there is a single pool, or a small number of larger pools is preferable since larger pools are less likely to be emptied.

Multiprocess with Devices Segregated by Process

Some application architectures may work best when a board’s channels are divided among a number of processes. This is possible if certain rules are followed and some pitfalls avoided.

The safest approach is to insure that each process has a unique set of devices to manipulate with no overlap of devices between processes. However, it is important to be careful that this strategy does not become what is, in effect, the older single process per channel model, which is grossly inefficient and does not scale well.

“Passing” a device from one process to another (for example, opening and closing a device in one process and then opening the same device in another process) should be avoided for two reasons.

- The application can lack clear logical design and become confusing.
- Device opening under the DM3 architecture is asynchronous by nature. Although device opening and closing appears to be a synchronous API call, a small

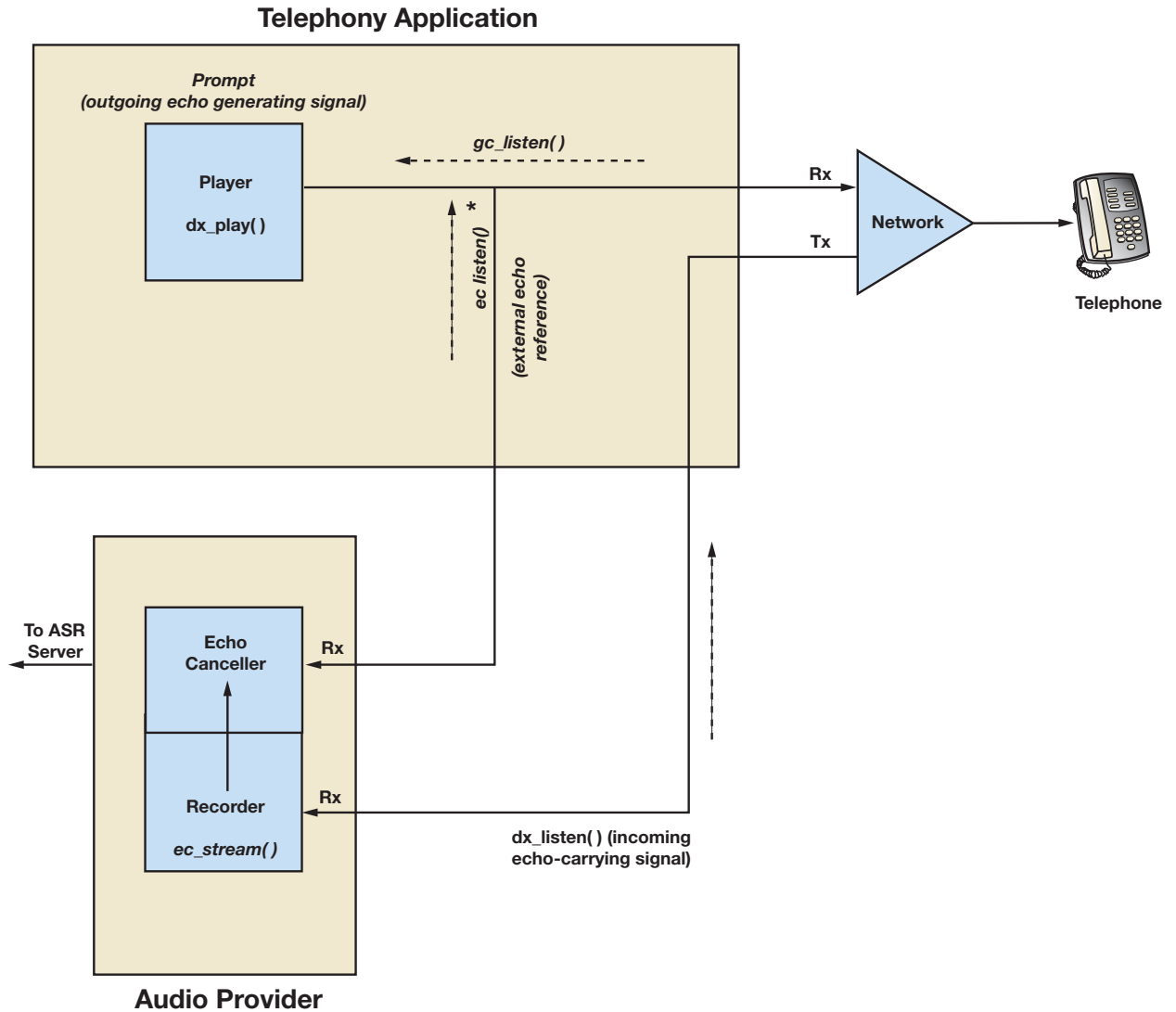


Figure 5. Sharing Resources in “Non-Telephony” Mode

Using the Nuance Audio Provider

The Nuance audio provider can be used in several ways when building telephony applications. For applications with simple telephony requirements, the “full control” scenario is best. However, when more complex functionality is required, “non-telephony mode” should be used.

Full Control

In “full control” mode, telephony, prompting, ASR, and TTS are all done within the audio provider and with the audio provider’s API controlling each of the functions. Intel R4 and Global Call APIs are never directly invoked from the application.

“Full control” mode is a very simple and straightforward way to use the audio provider with DM3 architecture. All network and media access is done in a proven manner internal to the audio provider, and this mode has

undergone extensive qualification, testing, and debugging and is supported by Nuance and Intel.

Non-Telephony Mode

“Non-telephony” mode allows another application in another process to provide telephony functionality. In this mode, prompting can be implemented through either the telephony application or the audio provider, but is more commonly done by the application. The audio provider does have some interaction with network and media channels that are opened and used by the application process.

Figure 5 illustrates how various telephony application components and the Nuance audio provider interact to safely share the resources necessary for an ASR session that uses DM3 architecture.

In Figure 5, the telephony application is handling call

control via the Global Call API and is prompting for each recognition. Since echo cancellation is an integral part of continuous speech processing (CSP), an Intel speech technology used in DM3 architecture, a reference signal must be supplied to the echo canceller that is controlled by the audio provider. This reference signal is only available in the telephony process that has initiated the `dx_play`, and this can only be done with a cross-process call (`ec_listen`) on a single channel's resources. The audio provider must monitor the transmit timeslot for the network device and ship the speech data found there to the recognition engine, necessitating another cross-process call. However, this cross-process channel sharing is done in a limited way that has been well designed and tested extensively.

Detailed instructions for sharing voice resources between the audio provider and a telephony application in non-telephony mode can be found in the *Nuance Speech Recognition System Version 8.5 Telephony Guide*.

Sharing CSP Voice Resources

The Nuance audio provider allows a telephony application to share voice resources on boards with DM3 architecture and CSP. A third-party application handles the telephony functions while the Nuance audio provider handles the recognition recording. Playback can be performed either by the third-party application or the audio provider. Sharing of CSP resources is needed in non-telephony mode to get full port density on the board.

To enable resource sharing:

- Set `audio.dialogic.CSPShareResource=TRUE`.
- Set `audio.Device` to the appropriate resource name.

To share voice resources on boards with CSP, the following rules must be observed:

- The third-party application and the Nuance audio provider must use the CSP APIs, especially the `ec_stopch` and `ec_stream` functions.
- The third-party application and the audio provider cannot call the function `dx_stopch`.
- The telephony application and the audio provider must be in separate processes.
- The telephony application must handle all of the timeslot setup. The audio provider must not be involved in assigning timeslots.
- The telephony application must set the appropriate event masks using function `dx_setevtmsk`. The audio provider must not call this function since the settings are shared between the application and the audio provider.

- DTMF events and playback can be handled by either the telephony application or the audio provider, but not by both.
- When using the audio provider in non-telephony mode, CT Bus connections must be re-established between the sound and telephony devices to enable recording. The parameter `audio.dialogic.TelephonyDevice` specifies the telephony front-end to which the sound device should be connected at initialization. The parameter must be set to the name of the network device (e.g., `dtiB1T1`, `dtiB1T2`) from which the recording should be made. If the parameter is not specified, a one-to-one match will be used (`dxxB1C1` will be connected to `dtiB1T1`, `dxxB1C2` to `dtiB1T2`, etc.).
- Before setting `audio.dialogic.CSPDevice` or calling `ec_setparm`, the application must poll the parameter `audio.dialogic.RecordState` after calling `StopRecording` to ascertain the recording state. The parameter `audio.dialogic.RecordState` returns a string indicating the current state. The possible values are:
 - `RECORDING` — Currently recording
 - `RECORD_STOPPING` — `StopRecording` has been called but recording has not yet stopped completely
 - `RECORD_PENDING` — `StartRecording` was called while an application was in the `RECORD_STOPPING` state
 - `RECORD_IDLE` — Recording has stopped. If the state is `RECORD_IDLE`, an application can set `audio.dialogic.CSPDevice` or call `ec_setparm`.

Dynamic vs. Static Device Allocation

Nuance recommends the use of static connections with its RCEngine API, the C++ API that is used to write an integration layer under an IVR's runtime environment. In the static model, each telephony port is associated with a single speech channel. The channel is instantiated during the initialization of the application and remains attached to its port until the application completes.

It is also possible for device assignments to be reset dynamically at runtime. This may be desirable if there are fewer CSP resources than voice resources in a system and the CSP resources are drawn from a pool and returned when required by the application. For example, if an audio provider instance in non-telephony mode is initialized with device `dxxxB1C1` (`audio.Device=dxxxB1C1`), a different device can be chosen at runtime by setting `audio.dialogic.CSPDevice=dxxxB1C2`.

For dynamic allocation, the parameter `audio.Device` must be specified with a single device only and not a range of devices.

You can enable dynamic device allocation by following these steps:

1. Set `audio.dialogic.CSPShareResource=TRUE`
2. Set `audio.dialogic.CSPDevice` to the new device. The audio provider will close the current device and reopen with the specified device.
3. Set `audio.dialogic.OffHook=TRUE`. The device is now ready for playback and recording.

See the *Nuance Platform Integrator's Guide, Version 8.5* for more information.

Moving from Springware to DM3 Architecture

As an example of how to re-architect an application with Nuance ASR and TTS components from using Springware to DM3 architecture, a single-process multithread architecture is described in this section.

Under Springware architecture, an application is divided among four processes

- **Media Manager** — Controls media functions for groups of channels: ASR prompting, play, record, and DTMF generation and detection
- **GlobalCall Call Control Manager** — Contains call-control-related functionality using the Global Call API
- **TTS Player** — Handles buffering and playing of audio data being returned from a TTS server
- **Nuance Audio Provider** — Only handles ASR streaming (non-telephony mode)

Vox devices are simultaneously opened in all processes. In the GlobalCall Manager, they are opened as part of the telephony channel for possible use in tone detection. The TTS Player uses multiple serial `dx_plays` to provide a steady stream of audio to the network Rx timeslot when TTS audio data is returned from a TTS server. The audio provider needs a `dx_open` and a `dx_listen` to a network transmit timeslot in order to obtain audio input that is streamed to the ASR engine.

While this approach is successful under the Springware architecture, sharing resources in this way under the DM3 architecture causes numerous delays in both the opening and use of audio devices. Even if it were possible to guard against simultaneous access on a

device, this arrangement would not work for applications using the DM3 architecture

A revised architecture is recommended. Combine the first three processes used in the Springware architecture into a single multithreaded process as described in the “Single Process” section earlier in this paper. Call control and media for all channels (with the exception of CSP-related media functions performed by the audio provider) will then be in the same application process.

TTS functionality may be improved by using the Streaming-to-Board API, available for the DM3 architecture in Intel Dialogic System Release 6.0 PCI for Windows. Correctly metering audio data into serial `dx_plays` can be very difficult, and the Streaming-to-Board API provides a buffering utility for throttling voice data from a TTS server to an open network channel that can be controlled more easily. The API functions allow the application programmer to create, maintain, and delete a circular stream buffer within a library, and they also provide notification when high and low watermarks are reached.

Identifying Multiprocess Problems

Intel engineers have observed several types of problems when multiple processes access the same channels in applications using DM3 architecture. These problems were solved by modifying and coordinating inter-process API calls. Note that these are mainly delay-related problems

The types of problems you may be able to anticipate are:

- Delays when responding to `dx_stopch()`
- Delayed or missing `TEC_STREAM` (`ec_stream()` termination) events
- Channel “hangs” and refuses to accept further calls
- Delays of 10 to 15 seconds in returning from `dx_close()`
- Unexpected “Device Busy” errors on shared devices
- Completion events not returned to the expected requester. For example, if two processes issue a `gc_WaitCall()` on the same device, the process to which the `GCEV_OFFERED` event is presented when the call arrives cannot be predicted.
- Gaps in audio due to voice buffer starvation. Although this problem is due to CPU loading approaching 100%, it has the appearance of a delay-oriented problem.

An Annotated Bibliography of Reference Information about the DM3 Architecture

This bibliography provides a list of documents that contain reference information about the DM3 architecture and related technologies such as the SRL and CSP. Annotations are included that explain the type of information that can be found in the documents.

Application Notes

Intel® Dialogic® System Release 6.0 on PCI for Windows*: The Standard Runtime Library—Optimizing Performance

<http://www.intel.com/network/csp/applnots/9029an.htm>

This application note explains the SRL in detail and compares the ways in which the Springware and DM3 architectures use the SRL.

T-1/ISDN Call Progress Analysis Using Global Call API on Intel® Telecom Boards

<http://www.intel.com/network/csp/applnots/8775FN01.htm>

This application note presents two methods for using products based on the DM3 architecture in identifying call progress analysis (CPA) indications: dx_dial and MakeCall.

Using Intel® Dialogic® DM3 Hardware for Creating Conference Bridge over IP or Telephony Network

<http://www.intel.com/network/csp/applnots/an03001.htm>

ConfDemo demonstrates enhanced DCB features for the DM3 architecture.

Intel® Dialogic® DMV160LP Products - Springware to DM3 Porting Guide on Windows* Operating Systems

<http://www.intel.com/network/csp/applnots/8695FN01.htm>

This application note provides developers using Intel Dialogic DMV160LP products in a Windows environment with a tool to simplify the process of porting existing enterprise messaging applications from Springware to DM3 architecture.

Using Optional ISDN Information Elements under the DM3 Architecture

<http://resource.intel.com/telecom/support/applnots/8892/index.htm>

This application note fills in the details of working with optional IEs that are not described in standard Global Call documentation. This technique is typically required for implementation of many ISDN supplementary services.

Intel NetStructure® DM/V1200A: Call Control Performance Testing

<http://www.intel.com/network/csp/applnots/8366FN01.htm>

This application note provides an example of a performance test conducted on a board with DM3 architecture.

Technical Notes

Running Continuous Speech Processing Using An External Reference Signal (ec_listen())

http://resource.intel.com/telecom/support/tnotes/gentnote/dl_soft/tn254.htm

The guidelines provided in the *Technical Discussion* section can be used to achieve up to 120 channels of CSP on a single Intel NetStructure board where the echo-reference signal comes from an external source rather than from an internal player (that is, the channel playing the prompt). This external source may reside in another process.

DMV/A Application Considerations when Multiple Devices Are Listening to the Same CT Bus Timeslot

<http://resource.intel.com/telecom/support/tnotes/tnbyos/2000/tn043.htm>

DMV, DMV/A, IPLink, VFN, and fax boards with network interfaces use the DM3 architecture and can route up to 256 (receive) timeslots externally to the CT Bus. As this can be a limitation in certain applications, a “Sharing of Timeslot” (SOT) algorithm is used to make efficient use of the number of timeslots externally routed to the CT Bus. This application note outlines the routing rules that applications must follow to allow this algorithm to work properly.

Intel Documentation

System Release Documentation Relating to DM3 Architecture

This section points to information about the DM3 architecture in the documentation for Intel Dialogic System Release 6.0 PCI for Windows. Scanning the information should help application programmers decide which API documentation to check more closely for information about the DM3 architecture.

Voice API Programming Guide

http://resource.intel.com/telecom/support/releases/winnt/sr60pci/onldoc/htmlfiles/voice_programming.html

6.2 Fixed and Flexible Routing Configurations — Discusses the use of SCbus routing of voice resource vs. hard assignments of voice to network resources

6.3 Fixed Routing Configuration Restrictions — Applicable to DM3 architecture

6.4 Additional DM3 Considerations

- Call Control through Global Call API Library
- Multithreading and Multiprocessing
- DM3 Interoperability
- DM3 Media Loads
- Device Discovery for DM3 and Springware
- Device Initialization Hint
- TDM Bus Time Slot Considerations
- Tone Detection

Continuous Speech Processing API Programming Guide

http://resource.intel.com/telecom/support/releases/winnt/sr60pci/onldoc/htmlfiles/csp_programming.html

1.1 Key Features — Overview of CSP features that differ between DM3 and Springware architectures

1.3.2 Tap Length — Echo cancellation differences between DM3 and Springware architectures

1.3.3 Adaptation Modes — Adaptation Mode support on DM3 vs. Springware architectures

1.9 Streaming to the TDM Bus — Enabling CSP bus echo-cancelled data streaming

1.10.1 Supported Data Formats on DM3 Boards — Audio data formats for DM3 vs. formats for Springware architectures

1.12 CSP Support on Springware Versus DM3 Boards — Table summarizing all CSP differences between DM3 and Springware architectures

4.1.2 Reserving Extra Time Slots for Streaming to TDM Bus — DM3 architecture considerations

4.1.3 Opening a Voice Channel — DM3 architecture and flexible routing

4.1.9 Setting Up Streaming or Recording — Automatic Gain Control differences between Springware and DM3 architecture

4.2.3 ISDN — CSP and ISDN use with Springware and DM3 architectures

5.2 VAD Operation — Voice Activity Detect differences between DM3 and Springware architectures

5.3 Fine-Tuning VAD Performance — Differences between DM3 and Springware architectures

6.1 Types of Buffers — Tables differentiate speech transfer buffers between DM3 and Springware architectures

6.2 Data Flow — Diagrams differentiate firmware to application data flow between DM3 and Springware architectures

6.3 Buffer Usage Tips — Tips for using DM3 vs. Springware architectures

Continuous Speech Processing API for Linux and Windows Operating Systems Library Reference

http://resource.intel.com/telecom/support/releases/winnt/sr60pci/onldoc/htmlfiles/csp_api.html

2.1 ec_getparm — Must run in same process as ec_setparm for DM3 architecture

2.1 ec_getxmitslot — Supports streaming echo-cancelled data to the TDM bus on select DM3 boards only

2.1 ec_reciottdata — DV_TPT terminating conditions supported on DM3 architecture

2.1 ec_setparm — List of all CSP parameters and their applicability to Springware and/or DM3 architectures

2.1 ec_stream — DV_TPT terminating conditions supported on DM3 architecture

Continuous Speech Processing API for Linux and Windows Operating Systems Demo Guide

http://resource.intel.com/telecom/support/releases/winnt/sr60pci/onldoc/htmlfiles/csp_demo.html

3 Preparing to Run the Demo – Running CSP demos in a system with mixed hardware, that is, some hardware using DM3 architecture and some using Springware architecture

Global Call API for Windows Programming Guide

http://resource.intel.com/telecom/support/releases/winnt/sr60pci/onldoc/htmlfiles/globalcall_programming.html

1.6 Global Call API versus DTI API – DM3 architecture considerations

1.7 Global Call API versus ISDN API – DM3 architecture considerations

3.4.1.3 Call Detection – GCEV_DETECTED event with DM3 architecture

3.4.1.4 Call Offered – GCEV_DETECTED event with DM3 architecture

3.4.1.10 Abandoned Calls – GCEV_DETECTED event with DM3 architecture

6.3 Using Global Call with DM3 Boards

- Routing Configurations Overview
- Working with Flexible Routing Configurations
- Working with Fixed Routing Configurations
- Handling Multiple Call Objects Per Channel in a Glare Condition

7.2 Call Progress Analysis when Using DM3 Boards

- Call Progress Analysis Definition
- Configuring Default Call Progress Analysis Parameters
- Configuring Call Progress Analysis on a Per Call Basis
- Setting Call Analysis Attributes on a Per Call Basis
- Configuring Call Progress Analysis on a Per Channel Basis
- Setting Call Analysis Attributes on a Per Channel Basis
- Customizing Call Progress Tones on a Per Board Basis

8.2.1 Generation of Events for Blocking Alarms – Alarm levels in DM3 architecture

Global Call API for Linux and Windows Operating Systems Library Reference

http://resource.intel.com/telecom/support/releases/winnt/sr60pci/onldoc/htmlfiles/globalcall_api.html

1 Function Summary by Category – Global Call API call support listed by network technology and applicability to DM3 or Springware architecture

2 gc_AttachResource – Restrictions on attaching network and voice resources with DM3 and Springware architectures

2 gc_GetCallState – State value support on boards with DM3 architecture

2 gc_MakeCall – Support of timeout interval with DM3 architecture

2 gc_OpenEx – Protocol_name field applicability to DM3 architecture

2 gc_OpenEx – Using boards with DM3 architecture in a fixed routing configuration

4 CT_DEVINFO – CT_DEVINFO struct fields used with DM3 architecture

4 DX_CAP – DX_CAP struct fields used with DM3 architecture

6.1 Alarm Source Object IDs – Differences between DM3 and Springware architectures

Nuance Documentation

This section contains references in Nuance documentation to DM3 architecture relating to installation, configuration, and use.

Nuance Voice Platform Telephony Guide

Nuance Voice Platform documentation is available with Nuance Voice Platform installation.

This guide describes the use of audio providers, including the Nuance audio provider for DM3 architecture, with the Nuance Voice Platform. Installing, configuring, testing, and troubleshooting are discussed.

Nuance Speech Recognition System 8.5 Telephony Guide

http://support.nuance.com/developers/documentation/index.html#nuance_base_8_5

This guide describes the features and usage model for the Nuance audio providers shipped with the Nuance System, including the audio provider for DM3 architecture. This guide is intended for application developers who want to add speech recognition to telephony applications.

To learn more, visit our site on the World Wide Web at <http://www.intel.com>.

1515 Route Ten
Parsippany, NJ 07054
Phone: 1-973-993-3000

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference <http://www.intel.com/performance/resources/Limits.htm> or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Intel, Intel Dialogic, Intel NetStructure, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

