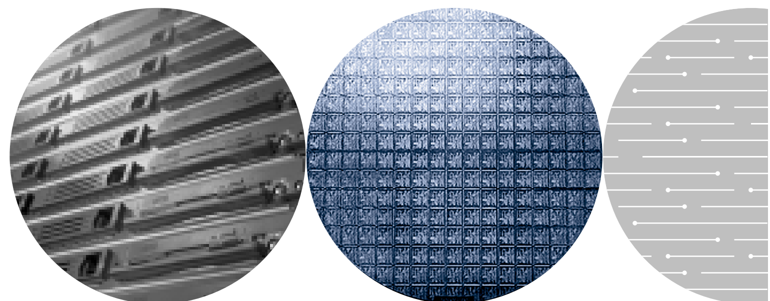
Application Note

A Solution to Invoke Synchronous Intel® Dialogic® Functions Asynchronously for the Windows* Operating System

Intel in Communications



Contents

Executive Summary 1

Introduction 1

The Issue 1

The Solution 2

Thread Pools 2

Why Use Thread Pools 3

Windows* Thread Pool Services 3

Windows Thread Pool Services Peculiarities 3

Handling Synchronous Functions 3

Notification to Application 5

Platform Independence 7

Application Example 7

Application Scenario 7

Application Initialization 7

Invocation of the Synchronous Function 9

Asynchronous Invocation of the Synchronous Function 9

Application Standard Runtime Event Loop 10

Conclusion 11

Figures

Figure 1. Synchronous and Asynchronous Modes of Operation 2

Executive Summary

This application note presents a solution to invoke Intel® Dialogic® synchronous functions asynchronously on Windows* operating systems. After a discussion on threads and thread pools, a proposed solution is implemented on a conferencing application.

Introduction

The solution presented in this document may be generically applied for any time-consuming function that blocks the invoking thread until its completion. It utilizes the Standard Run-time Library (SRL) from Intel in a multithreaded application environment for events management. Please note that any event management mechanism could be used instead of the SRL, but SRL provides a standard way of handing events in Intel® systems.

For more information on SRL, please refer to *Voice Software Reference: Standard Runtime Library for Windows* at http://resource.intel.com/telecom/support/releases/winnt/SR511/docs/htmlfiles/srlgd3/1458-02.htm.

Please note that readers of this application note are advised to test for multithread safety of the synchronous functions before attempting the solution presented here. The solution presented in this application note will only work for thread-safe synchronous functions.

The Issue

Most of the functions in Intel Dialogic functions allow the caller to select a function invocation mode, namely `EV_SYNC` and `EV_ASYNC` for synchronously or asynchronously invoking these functions, respectively. But some functions only provide a synchronous interface. Synchronous functions block the invoking thread or the entire application in single-threaded applications for the duration that the function executes (i.e., until it returns). Such a limitation is of particular concern to application developers, especially in high-density systems when some of these synchronous functions are time consuming. Such time-consuming functions limit the application from continuing to process its logic while waiting for the synchronous function to return.

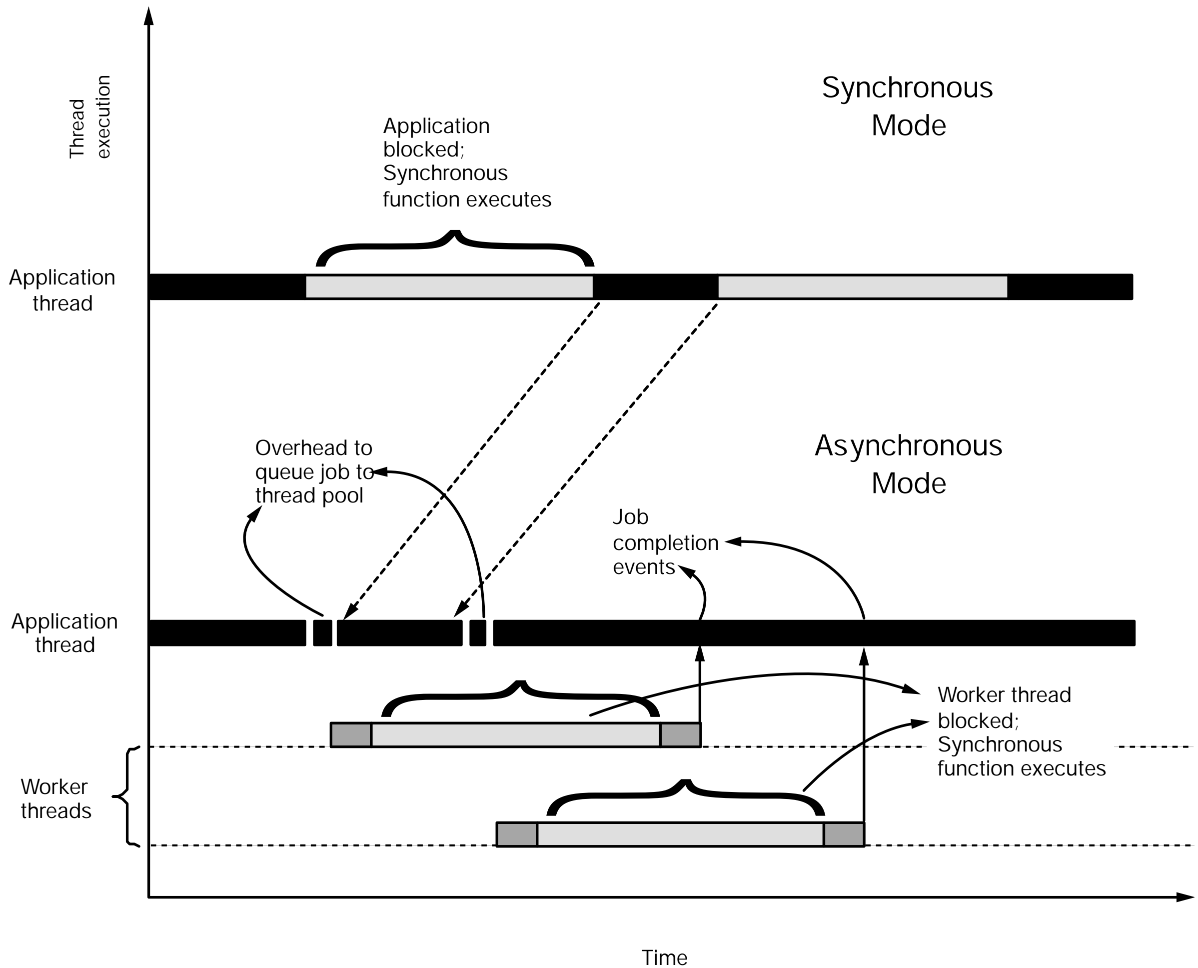


Figure 1. Synchronous and Asynchronous Modes of Operation

The Solution

Synchronous functions could be made asynchronous using multithreading. By spawning a new thread or using an already created thread for this purpose and transferring the responsibility of invoking the time-consuming synchronous functions to this "worker thread", the application or the calling thread could continue processing the logic while the worker thread waits for the synchronous function to complete its execution and return. When the synchronous function returns, the worker thread then posts an SRL event (explained later) to the application thread with appropriate return code as was returned by the synchronous function.

One may want to pre-create a pool of worker threads deemed to do the waiting job on behalf of the main application. That way the overhead associated in creation (and later destruction) of the "dynamic threads" could be circumvented. In such a scenario, one of the threads in the pool might do the business of scheduling jobs to an available thread in the pool.

The following figures explain the functioning and the benefits of this solution.

Thread Pools

This section presents a discussion on the usage of threads for the purpose of implementing the solution proposed in this application note.

Why Use Thread Pools

As explained before, by using an additional thread, one may relieve the main application thread of the task of waiting for the synchronous function. In other words, by using an additional thread, the application could continue processing its logic, without getting blocked, while the synchronous function is executed.

But if this additional thread were to be created at the time of the invocation of the synchronous function, the application spends the amount of time as is needed by the operating system to create this thread. A better solution would be to pre-create a set of worker threads at the application initialization time which would invoke any such synchronous functions as needed by the application. This pool of worker threads needs to be managed and "someone" should assign jobs to them with suitable load balancing. It is recommended to have a separate thread do the job assignment to these worker threads. So for a pool of "n" worker threads, one may assign the first thread to be the master thread or the control thread while the rest of the "n-1" threads, called the worker threads, do the job of invoking the synchronous function.

Windows Thread Pool Services*

Windows* provides a thread pooling mechanism natively and manages it in the operating system itself. It creates and deletes threads dynamically as needed by the application. The number of threads created is limited only by the available system memory.

The Windows function `QueueUserWorkItem` is used by the application to seek the services of the thread pooling library. Windows creates the thread pool the first time the application invokes this function. At least one thread in the pool is reserved to monitor the remaining threads in the pool and to assign them "work items" as requested by the application.

For further information, please refer to http://msdn.microsoft.com/library and search for `QueueUserWorkItem`.

Windows Thread Pool Services Peculiarities

Please note that the declaration of the function `QueueUserWorkItem` is not provided in the version of "`winbase.h`" supplied with Microsoft Visual Studio* 6.0. The developer should either manually declare the function as shown here, or include the Windows SDK version of "`winbase.h`" before including the Visual Studio headers.

```
extern "C" WINBASEAPI BOOL WINAPI QueueUserWorkItem(
  LPTHREAD_START_ROUTINE Function,
  PVOID Context,
  ULONG Flags
);
```

Handling Synchronous Functions

Any synchronous function is invoked asynchronously by developing a look-alike function that shares the same signature as shown here.

Synchronous function:

```
long foo(Type1 arg1, Type2 arg2 …, Typen argn)
```

Asynchronous version of the synchronous function:

```
long ASYNC_foo(Type1 arg1, Type2 arg2 …, Typen argn);
```

An associated structure is defined as shown here that wraps up the arguments of the function to pass to the thread pool. Please note that if any of the formal arguments is an address to a memory location (i.e., a pointer type variable), care should be taken that the memory pointed to by the pointer has a life at least until the time that the synchronous function is actually executed in the worker thread.

```
typedef struct tagFOO {
   Type1 arg1;
   Type2 arg2;
   .
.
.
   Typen argn;
} DFOO, *PFOO;
```

Now the application invokes the Intel function as shown here.

```
// asynchronous invocation
if (invokeAsynchronously)
{
    rc = ASYNC_foo(arg1, arg2, … , argn);
}
// synchronous invocation
else
{
    rc = foo(arg1, arg2, … , argn);
}
```

The `ASYNC_foo` is implemented as follows.

```
long WINAPI ASYNC_foo(Type1 arg1, Type2 arg2,
                                …, Typen argn)
{
   DFOO vContext = {
         arg1,
         arg2,
         .
         .
         .
         argn
   };
   PFOO pvContext = new DFOO (vContext);
   QueueUserWorkItem(THREAD_ASYNC_foo, pvContext,
                         WT_EXECUTEINLONGTHREAD);
   return 0;
}
```

Note that the `THREAD_ASYNC_foo` is the callback function that will be invoked in the context of the worker thread selected by the Windows thread pool service.

The callback function is further implemented as shown here.

```
#define ASYNC_EVENT_FOO 0x10000
void THREAD_ASYNC_foo(void* pvContext)
{
    long rc = 0;
    PFOO pContext = (PFOO)pvContext;
    rc = foo(pContext->arg1, pContext->arg2,
                 …, pContext->argn);
    // the synchronous function has returned;
    // post an event to the appln
    sr_putevt(devh, ASYNC_EVENT_FOO,
              sizeof(rc), &rc, 0);
    delete pContext;
}
```

The `ASYNC_EVENT_FOO` is the SRL user event that is posted to the application upon completion of the synchronous function, `foo`. Note that `foo` is actually indeed invoked synchronously, but now only the selected thread in the thread pool is the one waiting for its completion while the application is free to continue processing its logic instead of waiting for it to return.

Please note that the formal arguments are packed in a structure whose memory is shown to be allocated and freed dynamically in the code snippet shown in the next section. In the case of high-density systems, one may use buffer pools to have pre-allocated memory to avoid the overheads of such dynamic memory allocation and freeing.

Notification to Application

The completion of the synchronous function invocation is notified to the application by the worker thread using an event posting mechanism provided by SRL. Please note that other means of event management may also be used to achieve the objective.

The major function of the SRL is to provide a common interface for event handling and other functionality common to all Intel devices. The SRL serves as the centralized dispatcher for events that occur on all Intel devices. Through the SRL, events are handled in a standard manner.

Upon completion of the synchronous function, the `THREAD_ASYNC_foo` callback function posts a predefined event to the application using the SRL function `sr_putevt`. The event is posted by the callback function as shown here.

```
// return code
long rc;
rc = foo(…);
sr_putevt(devh, ASYNC_EVENT_FOO,
          sizeof(rc), &rc, 0);
```

The application in turn waits for an SRL event using one of the SRL wait functions, for example, `sr_waitevt` as shown below. The function `sr_waitevt` waits for any event until a specified timeout period.

```
// loop infinitely
while (1)
{
    int ret;
    ret = sr_waitevt(-1 /* wait forever */);
    if (ret != -1)
    {
        ProcessEvents();
    }
}
```

The function `ProcessEvents` is the applications event handler. All events received in the application's SRL event loop is processed in this function. This function merely analyses the received event and performs the event-specific processing.

```
void ProcessEvents()
{
    long ev;
    long dev;
    void* vp;

    ev  = sr_getevttype(0);
    dev = sr_getevtdev(0);

    vp = sr_getevtdatap(0);

    switch (ev)
    {
    case ASYNC_EVENT_FOO:
        {
            long rc;

            if (rc != 0)
            {
                printf("Error in Device = %d\n", dev);
                return;
            }

            rc = (long*)vp;
            .
            .
            .
        }
        break;
    default:
        {
            .
            .
            .
        }
        break;
    }
}
```

Platform Independence

The solution proposed in this application note is presented in a Windows-specific implementation. One may develop a platform-independent thread pool library which the application may use regardless of the underlying operating system. In case of Windows, such a library may utilize Window's native thread pooling services while in the case of other operating systems, it may provide a custom implementation of the thread pooling mechanism.

Application Example

The solution proposed in this application note is exemplified using a conferencing application. The application takes two arguments.

```
mode: 0 = Synchronous; 1 = Asynchronous
# of parties in conference
```

Application Scenario

The application creates one conference with the user-supplied number of parties in it. The first party plays a file and the rest of the parties record the file. The asynchronous invocation is demonstrated when the application is run in the asynchronous mode. The functions `dcb_estconf` and `dcb_addtoconf` exemplify the solution. These two functions are part of the R4 API from Intel and only provide a synchronous mode of invocation. But the application illustrates how a developer may invoke these functions in the asynchronous mode.

Application Initialization

```
int main(int argc, char* argv[])
{
    int   iIndex      = 0;
    short iIterator   = 0;    // to access global arrays

    // collect command line parameters
    .
    .
    .

    if (async)
    {
        // initiale thread pool library
        .
        .
        .
    }
    else
    {
        .
        .
        .
    }

    // Perform Application specific initializations
    .
    .
    .
```

```
    // Open all voice resources
    .
    .
    .

    // Open a conference resource
    .
    .
    .

    // Put up to 4 parties in conference
    iIterator = 0;
    if(-1 == EstablishConf(iIterator,
        ((nConfItems <= 4) ? nConfItems : 4)))
    {
        printf("dcb_estconf for Host failed\n");
    }

    // Start Infinite App Main Loop
    if (async)
    {
        while (1)
        {
            int ret;
            /* wait forever */
            ret = sr_waitevt(-1);
            if (ret != -1)
            {
                ProcessEvents();
            }
        }
    }
    // Note: In async mode, we'll never reach here
    //        Following code only executes in sync mode


    // Put everyone else in the Conference
    if (nConfItems > 4)
    {
        AddToConf(4, nConfItems-4));
    }

    StartConferencing();

    .
    .
    .
}
```

Invocation of the Synchronous Function

The asynchronous versions of each synchronous function `foo` would be `ASYNC_foo`. Both the functions would share the same function signature.

```
    if (async)
     {
         rc = ASYNC_dcb_estconf(g_arpConfDevice,
            &g_arpCDT[ a_sCounter] ,
            a_sItemsInConference, MSCA_ND,
            &g_arpConfID);
     }
     else
     {
         rc = dcb_estconf(g_arpConfDevice,
            &g_arpCDT[ a_sCounter] ,
            a_sItemsInConference, MSCA_ND,
            &g_arpConfID);
    }
```

Note, in case of asynchronous invocation, the return code will only indicate that the delegation of this work item has been successfully posted to a worker thread. The actual return error code will be available when the synchronous function completes execution in the context of the worker thread, which will be posted to the applications event loop by the worker thread.

Asynchronous Invocation of the Synchronous Function

In order to delegate the responsibility of actually invoking the synchronous function by a worker thread, the asynchronous version of the function must pack the function arguments into a structure, allocate memory for this structure, and then post a "user work item" to the Windows' thread pool library using the Windows API, `QueueUserWorkItem`.

```
  typedef struct tagDCB_ESTCONF {
     int      devh;
     MS_CDT*  cdt;
     int      numpty;
     int      confattr;
     int*     confid;
  }  DCB_ESTCONF, *PDCB_ESTCONF;

  long WINAPI ASYNC_dcb_estconf(int devh, MS_CDT* cdt, int numpty,
                                int confattr, int* confid)
  {
      DCB_ESTCONF vContext =
      {
          devh,
          cdt,
          numpty,
          confattr,
          confid
      };
      PDCB_ESTCONF pvContext = new DCB_ESTCONF(vContext);

      QueueUserWorkItem(THREAD_ASYNC_dcb_estconf, pvContext,
```

```
                 WT_EXECUTEINLONGTHREAD);
    return 0;
}

void THREAD_ASYNC_dcb_estconf(void* pvContext)
{
    long rc = 0;
    PDCB_ESTCONF pContext = (PDCB_ESTCONF)pvContext;

    rc = dcb_estconf(pContext->devh, pContext->cdt,
                       pContext->numpty, pContext->confattr,
                       pContext->confid);
    sr_putevt(pContext->devh, DCB_EVENT_ESTCONF,
                 sizeof(rc), &rc, 0);
    delete pContext;
}
```

Application Standard Runtime Event Loop

The following code segment provides a sample event handler implementation for the conferencing application discussed in this application note.

```
void ProcessEvents()
{
    long ev;
    void* vp;

    ev = sr_getevttype(0);
    vp = sr_getevtdatap(0);

    switch (ev)
    {
    case DCB_EVENT_ESTCONF:
        {
            long rc;
            rc = (long*)vp;

            if (rc != 0)
            {
                printf("Error\n");
                return;
            }

            // Put everyone else in the Conference
            if (nConfItems > 4)
            {
                AddToConf(4, nConfItems-4);
            }
            else
            {
                StartConferencing();
            }
```

```
            }
            break;
        case DCB_EVENT_ADDTOCONF:
            {
                long rc;
                rc = (long*)vp;

                if (rc != 0)
                {
                    printf("Error\n");
                    return;
                }

                static int partiesInConf = 4;
                if (++partiesInConf == nConfItems)
                {
                    StartConferencing();
                }
            }
            break;
        .
        .
        .
        default:
            {
            }
            break;
    }
}
```

Conclusion

In a simple test run of the conferencing sample application with 32 parties in a conference, the synchronous mode of invocation was found to block the application for 3545 ms, whereas the asynchronous wrapper only blocked the application for 421 ms. In other words, the solution presented in this application note helped the application process the synchronous function in about 12% of the time it took to invoke in the synchronous mode. In most applications this a significant amount of time in which other application logic could be performed. The test system specifications were as follows:

- System release: Intel Dialogic SR 5.1.1 FP1 for Windows
- Operating system: Windows 2000 SP 3
- Processor speed: 500 MHz
- RAM size: 128 MB
- Intel Dialogic boards: DM/V1200A-4E1, DM/V1200-4E1

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 applications.

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

Intel, Intel Dialogic, 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.

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/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Printed in the USA Copyright © 2003 Intel Corporation All rights reserved. Printed on recycled paper.

06/03 00-8710-001