

## Creating Services with C++Builder

by Randy Charles Morin

This will be a quick one. When I wrote my book on Services Windows, the new TService component for Delphi and C++Builder were new to the table. I didn't write about this service in the book, so I thought I should spend a few minutes discussing this component in an article for my readers. I wish I could somehow send this to each person who bought a copy of the book, but most people don't send their email addresses to me. I can't figure out why☺

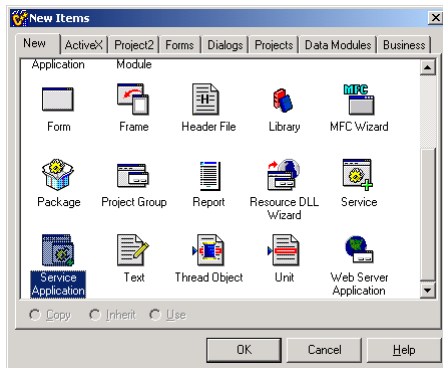
### Starting your Service

I've assumed throughout the article that you know how to start, stop, pause and continue services. In the case where you are not familiar with these concepts, let me direct you to the Services MMC snap-in. This MMC snap-in is typically available in the Start menu under Programs | Administrative Tools | Services.

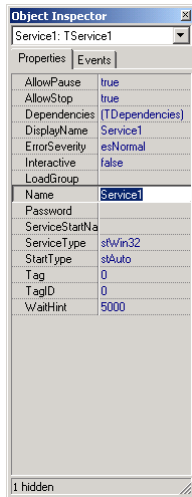
### The TService Component

Creating a Windows Service in C++Builder version 5.0 is pretty simple. From the menu bar, select File | New. Then select Service Application from the New Items dialog (see figure 1) and click OK.

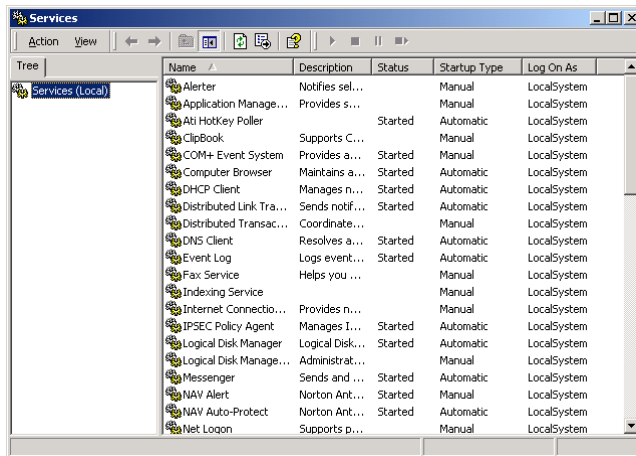
Figure 1: New Items Dialog



Tada! That's it. You have a full service application. You'll likely want to change some of the properties for your object. You can do this in the Object Inspector window (see figure 2).

**Figure 2: Object Inspector Window**

You'll may want to change the Name of the service. This is the class name for your TService derived class. The DisplayName property is the name that'll appear Services Control Panel applet in Windows NT and the Services MMC snap-in in Windows 2000 (see figure 3).

**Figure 3: Service MMC Snap-in**

The AllowPause and AllowStop properties enable and disable the ability to pause and stop the service from the Service snap-in. The StartType property can be changed to Manual if you don't want the service to automatically start when your Windows computer is started.

The code generated for your project is quite minimal. The project file (see Listing 1), the header file (see Listing 2) and the source file (see Listing 3) are shown.

**Listing 1: Project2.cpp**

```
#include <SysUtils.hpp>
#include <SvcMgr.hpp>
#pragma hdrstop
#define Application SvcMgr::Application
USERES("Project2.res");
USEFORM("Unit1.cpp", Service1); /* TService: File Type */
//-----
```

```

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TService1), &Service1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Sysutils::ShowException(&exception, Sysutils::ExceptAddr());
    }
    return 0;
}

```

Borland did a very good job with the project source. This source listing looks nearly identical to all other project types.

### Listing 2: Unit1.h

```

//-----
#ifndef Unit1H
#define Unit1H
//-----
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <SvcMgr.hpp>
#include <vcl.h>
//-----
class TService1 : public TService
{
__published:    // IDE-managed Components
private:       // User declarations
public:        // User declarations
    __fastcall TService1(TComponent* Owner);
    TServiceController __fastcall GetServiceController(void);

    friend void __stdcall ServiceController(unsigned CtrlCode);
};
//-----
extern PACKAGE TService1 *Service1;
//-----
#endif

```

You may add a few methods to your TService class, but generally it will have the form shown in Listing 2 and 3.

### Listing 3: Unit1.cpp

```

//-----
#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"

TService1 *Service1;
//-----
__fastcall TService1::TService1(TComponent* Owner)
    : TService(Owner)
{
}

TServiceController __fastcall TService1::GetServiceController(void)
{
    return (TServiceController) ServiceController;
}

void __stdcall ServiceController(unsigned CtrlCode)
{
    Service1->Controller(CtrlCode);
}

```

```
}
//-----
```

At this point, you have a service that can be started, stopped, paused, continued, installed and uninstalled. But your service doesn't do all that much. In order to add any more functionality, you have to override one of the event-handling messages. The most obvious method to override is the OnExecute method.

## Background Thread

You might consider adding COM or socket components. For my example, I'll just have a thread loop and beep every time through the loop.

Each TService has one background thread that can be overridden to implement background functionality in the service. Or at least that's what the document says. So I tried overriding the OnExecute method.

### Listing 4: TServiceThread::OnExecute

```
void __fastcall TMyService::ServiceExecute(TService *Sender)
{
    while (!Terminated)
    {
        ::Sleep(1000);
        ::Beep(37, 100);
    }
}
```

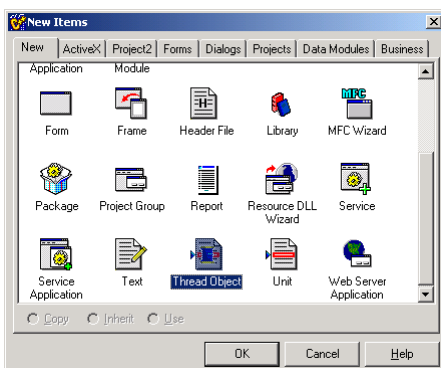
This didn't work. I tried variations on this theme, but found that no matter what I did, overriding the OnExecute method led to bad behaviors in the service. From this experimentation, I decided that you did not want to override the OnExecute method of the TService class.

### Lesson #1

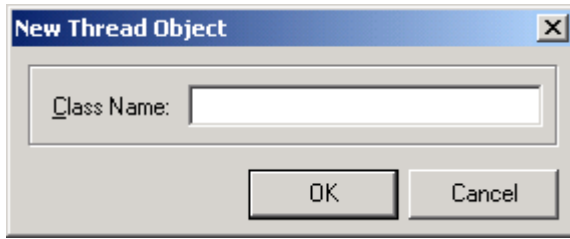
Do not override the OnExecute method of the TService class.

The alternative is to create your own background thread. I'll run you thru the steps of creating your own threads. Select File | New from the C++Builder menu bar.

Figure 4: New Items Dialog



This time select the Thread Object and click OK. You will be prompt for a class name.

**Figure 5: New Thread Object Dialog**

I used the class name MyThread. I made a couple of modifications to the MyThread class. I created a global pointer to an instance of the class and wrote the implementation to the Execute method. See Listing 5 and 6.

**Listing 5: Unit2.h**

```
//-----
#ifndef Unit2H
#define Unit2H
//-----
#include <Classes.hpp>
//-----
class MyThread : public TThread
{
private:
protected:
    void __fastcall Execute();
public:
    __fastcall MyThread(bool CreateSuspended);
};
//-----
extern MyThread * MyThread1;
#endif
```

I added the extern definition in the declaration file of the MyThread class. See Listing 5.

**Listing 6: Unit2.cpp**

```
//-----
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
#pragma package(smart_init)
//-----

// Important: Methods and properties of objects in VCL can only be
// used in a method called using Synchronize, for example:
//
//     Synchronize(UpdateCaption);
//
// where UpdateCaption could look like:
//
//     void __fastcall Unit2::UpdateCaption()
//     {
//         Form1->Caption = "Updated in a thread";
//     }
//-----

MyThread * MyThread1 = 0;

__fastcall MyThread::MyThread(bool CreateSuspended)
    : TThread(CreateSuspended)
{
}
```

```
//-----
void __fastcall MyThread::Execute()
{
    while (!Terminated)
    {
        ::Sleep(1000);
        ::Beep(37, 100);
    }
}
//-----
```

I added the MyThread1 pointer and the implementation of the Execute method in the source file of the MyThread class. See Listing 6.

## Shared Process Services

Many services share the same process. For example the services.exe process has a large number of services running within. The Alerter, Appmgmt, Browser, Dhcp, Dmserver, Dnscache, Eventlog, Lanmanserver, Lanmanworkstation, LmHosts, Messenger, PlugPlay, ProtectedStorage, Seclogon, TrkWks, W32Time and Wmi services all share this one process in Windows 2000 Professional. That's a lot of services in one process.

The reason they would share the same process is that if you created single process services for each, then you would end up with a very lengthy process list. Another benefit is that shared process services can also share in-process memory and resources. If all seventeen services in the services.exe process share one piece of memory, then your memory allocation is 94% more efficient than if they did not share that piece of memory. Efficiencies on that order can prove very valuable. On the other hand, memory is cheap.

Nonetheless, you may find either of the two reasons I have presented here or any number of other reasons to justify having two or more services in the same process. If you do decide that you want this, then you will encounter a big problem with the TService component. Although it is advertised to support more than one service per process, it simply does not. The reason is a small coding problem in the code-generating wizard.

If you look back at listing 3, then you'll see that the wizard generated a function called ServiceController. The wizard will generate a function of this same name for every service in your process. I don't consider myself an expert on the linking of functions with the same name, but the end result is that each TService derived class will share the same one instance of the function. Each service will start correctly, but only one of the services in the process will be capable of any other interactions with the service control manager. That is, you won't be able to pause, continue or stop any of the other services. Yikes!

But don't worry, I wouldn't be going on and on about this unless I had the solution. The solution is to rename the ServiceController function for each service instance. See Listing 7 and 8.

### Listing 7: Unit2.h

```
//-----
#ifndef Unit2H
#define Unit2H
//-----
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <SvcMgr.hpp>
#include <vcl.h>
```

```
//-----
class TService2 : public TService
{
    __published:    // IDE-managed Components
private:          // User declarations
public:           // User declarations
    __fastcall TService2(TComponent* Owner);
    TServiceController __fastcall GetServiceController(void);

    friend void __stdcall ServiceController2(unsigned CtrlCode);
};
//-----
extern PACKAGE TService2 *Service2;
//-----
#endif
```

In listing 7 and 8, I changed each instance of the ServiceController function name to ServiceController2, in order to differentiate from other instances of this function.

**Listing 8: Unit2.cpp**

```
//-----
#include "Unit2.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"

TService2 *Service2;
//-----
__fastcall TService2::TService2(TComponent* Owner)
    : TService(Owner)
{
}

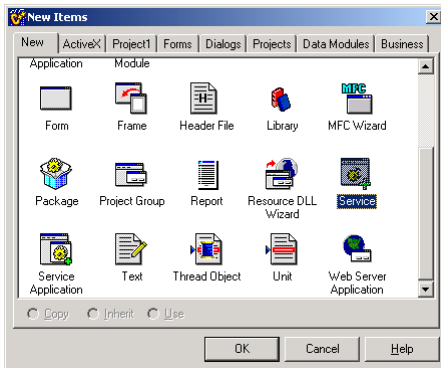
TServiceController __fastcall TService2::GetServiceController(void)
{
    return (TServiceController) ServiceController2;
}

void __stdcall ServiceController2(unsigned CtrlCode)
{
    Service2->Controller(CtrlCode);
}
//-----
```

After making these changes you can then add as many services as you want to your service executable and they will all share the same process. You will of course have to name the ServiceController function differently for each TService derived class in your project.

Now that I've described the problems and solutions of having multiple TService derived classes in one project, I'll show you how it is done. Go about create a Service Application project as I described in earlier sections of this article. Then, for each addition service beyond the first you must add one Service component to your project. You add the Service component by selecting File | New from the menu bar. See Figure 6.

**Figure 6: New Items Dialog**



Then from the New Items dialog, select Service and click the OK button. The wizard will generate a unit header and source with the declarations and definitions of one TService derived class per service added to your project.

## Installing your Service

After you have created a service executable, you must install it before you can start it. You can install your service by running the executable with the “/install” command-line. You’ll likely install your service, run it and find a bug. Then you’ll make some changes and try to install the service again.

Before installing your service a second time, you’ll be forced to uninstall it. You can uninstall your service by running the executable with the “/uninstall” command-line.

## Conclusion

The TService component in C++Builder can help a junior team of developers move quickly into developing Windows Services. But, I would like to point out that the TService component like most components are somewhere between minimal and complete. The TService component is less than complete as not all the functionality of services is available to the developer. The TService component is more than minimal as it requires a lot of understanding of the Service Control Manager API in order to understand the numerous pitfalls, bugs and limitations in the component.

If you are thinking about implementing anything beyond simple services, then you might find yourself constrained in the use of this component. You might be better off taking a day to see if you could write a better service framework using the Service Control Manager API. If you do decide to write your own framework, then let me suggest two things.

#1 - Buy a copy of my book Programming Windows Services or Kevin Miller’s book Professional NT Services. Actually, get both. Books are cheap. Expense them to your company.

#2 - If you are less than a senior C++ Win32 API developer, then don’t even thinking about writing your own framework.