



ON TARGET
software
www.targetsoft.com

C++ Notifiers
Programmer's Guide
Revision 1.2

Copyright 1998-2008 On Target Software All Rights Reserved



Table of Contents

PLEASE READ FIRST	3
C++ NOTIFIERS OVERVIEW	4
WHAT ARE NOTIFIERS?.....	4
HOW ARE THEY USED?	4
QUICK EXAMPLE	5
WITHOUT NOTIFIERS	5
WITH NOTIFIERS	7
HOW NOTIFIERS WORK.....	9
THE FOUR PLAYERS IN NOTIFICATION	9
NOTIFIER CLASS HIERARCHY	10
THE NOTIFICATION PROCESS	11
GETTING STARTED	12
BUILD ENVIRONMENT	12
DOWNLOAD AND INSTALL	12
BUILD THE THERMOSTAT SAMPLE	12
INTEGRATE NOTIFIERS INTO YOUR APPLICATION.....	13
IT'S NOT A PANACEA.....	13
INTEGRATION.....	14
1. CREATE THE NOTIFIER_CLASS ENUM.....	14
2. SUBCLASS CSUBSCRIBER	14
3. SUBCLASS CDISPATCHER	15
4. CREATE AND INITIALIZE THE DISPATCHER	15
ADDING A NOTIFIER CLASS	16
A. DEFINE THE NOTIFIER SUBCLASS.....	16
B. ADD TO THE NOTIFIER_CLASS ENUM	16
C. ADD A VIRTUAL FUNCTION TO THE CSUBSCRIBER CLASS.....	16
D. ADD A CASE STATEMENT TO <i>DISPATCHToONE</i>	16
USING NOTIFIERS	17
PUBLISHING.....	17
SUBSCRIBING.....	17
CNOTIFIER CLASS	18
OVERVIEW	18
PROPERTIES.....	18
METHODS.....	18
CSUBSCRIBER CLASS	19
OVERVIEW	19
PROPERTIES.....	19
METHODS.....	19
CDISPATCHER CLASS.....	20
OVERVIEW	20
PROPERTIES.....	20
METHODS.....	20
MORE FREE SOFTWARE.....	21
ERROR SIMULATION	21
DEBUG MESSAGE LOGGING.....	21
WHY ALL THIS FREE SOFTWARE?	22
HOW TO REACH ON TARGET SOFTWARE.....	22
REVISION HISTORY.....	22
1.0 TUESDAY, NOVEMBER 25, 1997.....	22
1.1 SUNDAY, JULY 19, 1998	22
1.2 THURSDAY, NOVEMBER 6, 2008	22



Please Read First

The documentation and source code is free for your use in any software you develop. You may not, however, copy the source files or documentation or any significant part of them without also including the copyright notice and web address for On Target Software in the source code and the source code documentation.

If others want a copy of the source code, please direct them to my website at www.targetsoft.com.

Neither On Target Software nor Dave Pomerantz assume any responsibility or liability for errors, incorrect results, or other problems that may occur with these files or with software in which these files are embedded. Use at your own risk.

If you run into problems with the source code or if you have questions, please send e-mail through the [Contact](#) link on my website. If you want to send attachments, contact me and I'll reply. That way I can keep my email address out of the view of spam robots.

The C++ Notifiers code is freely available with these simple restrictions. If you cannot abide by them, please don't use the software.

Thank you!



C++ Notifiers Overview

This is a programmer's guide and reference for the use of notifiers. It is intended to accompany the more general explanatory article found [here](#). This document explains how and why you might use notifiers, how to build the DLL and the thermostat sample, and how each notifier class and method is built and used.

What are notifiers?

Notifiers are a messaging protocol. They allow unrelated objects to communicate asynchronously. One object sends a notifier. Another object receives it. Neither object knows of the other; they share only a common notifier interface.

By providing a small set of notifiers, dozens of objects can communicate without knowing anything more than the details of the notifiers they're sending and receiving. System complexity is kept to a minimum, even in large systems.

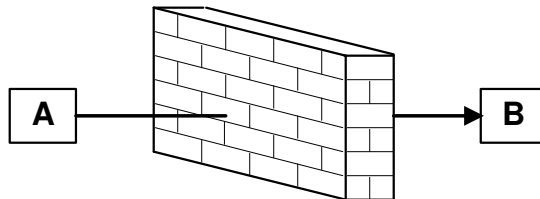
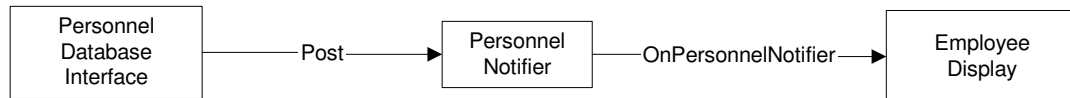


Figure 1. Notifiers allow objects to communicate anonymously.

How are they used?

A notifier class stores a piece of information: a font, a database record, a control setting. The data object responsible for that information creates and posts the notifier when the information changes. A dependent object that needs the information receives the notifier and updates itself. In this example, a personnel database object signals that someone's name has changed. The employee display redraws itself, but the personnel database and the employee display are not directly connected.





Quick Example

Let's say you're writing process automation software and you have a text window displaying the temperature in a reactor vessel. When the device driver for the temperature sensor detects a temperature change, how is the text window updated?

Without notifiers

In C++, without notifiers, the sensor object and the window object are connected with pointers. The sensor calls the window whenever the temperature changes. The window calls the sensor to get the temperature when the window first displays. Upon destruction, each object calls the other so neither is left with a dangling pointer.

As a result, both objects:

- carry pointers to each other
- include each other's header files
- are dependent on each other's existence

The sensor object, in C++, without notifiers

```
#include "text_window.h"
// Set pointer to window
void CSensor::SetWindowPointer(CTextWindow *pTextWindow)
{
    m_pTextWindow = pTextWindow;
}
// Tell window this sensor is destroyed
void CSensor::~CSensor()
{
    if (m_pTextWindow)
    {
        m_pTextWindow->OnDestroySensor(this);
    }
}
// Find out when window is destroyed
void CSensor::OnDestroyWindow()
{
    m_pTextWindow = 0;
}
// Periodically check the temperature
void CSensor::PollTemperature()
{
    float nTemperature = GetTemperature();
    if (abs(nTemperature - m_nTemperature) > 0.01)
    {
        m_nTemperature = nTemperature;
        if (m_pTextWindow)
        {
            m_pTextWindow->OnTemperatureChange(m_nTemperature);
        }
    }
}
// return current temperature
float CSensor::GetTemperature()
{
    return m_nTemperature;
}
```



The text display object, in C++, without notifiers

```
#include "sensor.h"
// Set pointer to sensor
void CTextWindow::SetSensorPointer(CSensor *pSensor)
{
    m_pSensor = pSensor;
}
// Tell sensor this window is destroyed
void CTextWindow::OnDestroy()
{
    if (m_pSensor)
    {
        m_pSensor->OnDestroyWindow(this);
    }
}
// Find out when sensor is destroyed
void CTextWindow::OnDestroySensor()
{
    m_pSensor = 0;
}
// Display initial value of temperature
void CTextWindow::OnInitialUpdate()
{
    if (m_pSensor)
    {
        DisplayTemperature(m_pSensor->GetTemperature());
    }
}
// Display change in temperature
void CTextWindow:: OnTemperatureChange(float nTemperature)
{
    DisplayTemperature(nTemperature);
}
```

Operation without notifiers

For two C++ objects to communicate without notifiers, they need to take the following steps:

- | | |
|-------------------|---|
| Exchange pointers | The owner object, perhaps the application object, creates the sensor and the text window, and gives each the other's pointer, using <i>SetWindowPointer</i> and <i>SetSensorPointer</i> . |
| Destroy pointers | Upon destruction, each object must tell the other that it's being destroyed, so neither object is left with a dangling pointer. |
| Get initial data | The window needs to call the sensor to retrieve the temperature |
| Send updates | The sensor calls the window's <i>OnTemperatureChange</i> function whenever the temperature has changed significantly. |



With notifiers

Using notifiers, neither object is aware of the existence of the other. They only know about the notifier. The sensor object posts the notifier when requested and when the temperature changes. The window object responds to the notifier.

As a result, the objects:

- do **not** carry pointers to each other
- do **not** include each other's header files
- are **not** dependent on each other's existence

They depend only on the notifier, which has the definition shown below.

The notifier class definition

```
// temperature notifier
class CTemperatureNotifier : public CNotifier
{
public:
    CTemperatureNotifier(void) { m_nTemperature = 0.0 };

    float m_nTemperature;
};
```

The sensor object, in C++, with notifiers

```
// Periodically check the temperature
void CSensor::PollTemperature()
{
    float nTemperature = GetTemperature();
    if (abs(nTemperature - m_nTemperature) > 0.01)
    {
        m_nTemperature = nTemperature;
        CTemperatureNotifier *pNotifier = new CTemperatureNotifier();
        if (pNotifier)
        {
            pNotifier->m_nTemperature = nTemperature;
            pNotifier->Post();
        }
    }
}

// Respond to request for temperature
void CTemperature::OnRequestNotifier(CRequestNotifier *pNotifier)
{
    if (pNotifier->m_nClass == TEMPERATURE_NOTIFIER)
    {
        CTemperatureNotifier *pNotifier = new CTemperatureNotifier();
        if (pNotifier)
        {
            pNotifier->m_nTemperature = m_nTemperature;
            pNotifier->Post();
        }
    }
}
```



The text display object, in C++, with notifiers

```
// Display initial value of temperature
void CTextWindow::OnInitialUpdate()
{
    Subscribe(TEMPERATURE_NOTIFIER);
    RequestNotifierPost(TEMPERATURE_NOTIFIER);
}
// Respond to change in temperature
void CTextDegrees::OnTemperatureNotifier(CTemperatureNotifier *pNotifier)
{
    DisplayTemperature(pNotifier->m_celsius);
}
```

Operation with notifiers

For two C++ objects to communicate with notifiers, they need to take the following steps:

- | | |
|-------------------|---|
| Subscribe | The window subscribes to the notifier. |
| Notify | The sensor posts a notifier whenever the temperature changes. |
| Respond to change | The window responds to the temperature change notifier. |



How Notifiers Work

The four players in notification

The publisher	is any object that creates a notifier and posts it to the dispatcher's queue.
The subscriber	is a CSubscriber object that tells the dispatcher which notifiers it wants to receive and responds to the notifiers when they arrive.
The notifier	is a CNotifier object that encapsulates information about a specific event.
The dispatcher	is a module in the notify.dll that tracks which subscribers want which notifiers, and delivers notifiers to the appropriate subscribers.

The Publisher

The publisher tells the system about a change. For example, the temperature sensor is a publisher.

The publisher can either post or send notifiers. Posting is asynchronous: the notifier is queued and the publisher continues to execute without waiting for the notifier to be dispatched. Sending is synchronous: the notifier is dispatched to all its subscribers before the publisher continues.

Publishers can also be subscribers.

The Notifier

The notifier encapsulates information about a change and contains its own copy of the data. This is important because a notifier stays in the dispatch queue after posting, so it may outlive the publisher and the original data.

The Subscriber

The subscriber is any object that inherits from the CSubscriber class.

Notifiers are dispatched to subscribers by calling a CSubscriber virtual function. For this reason, each notifier class has a corresponding virtual function in the CSubscriber class¹.

The Dispatcher

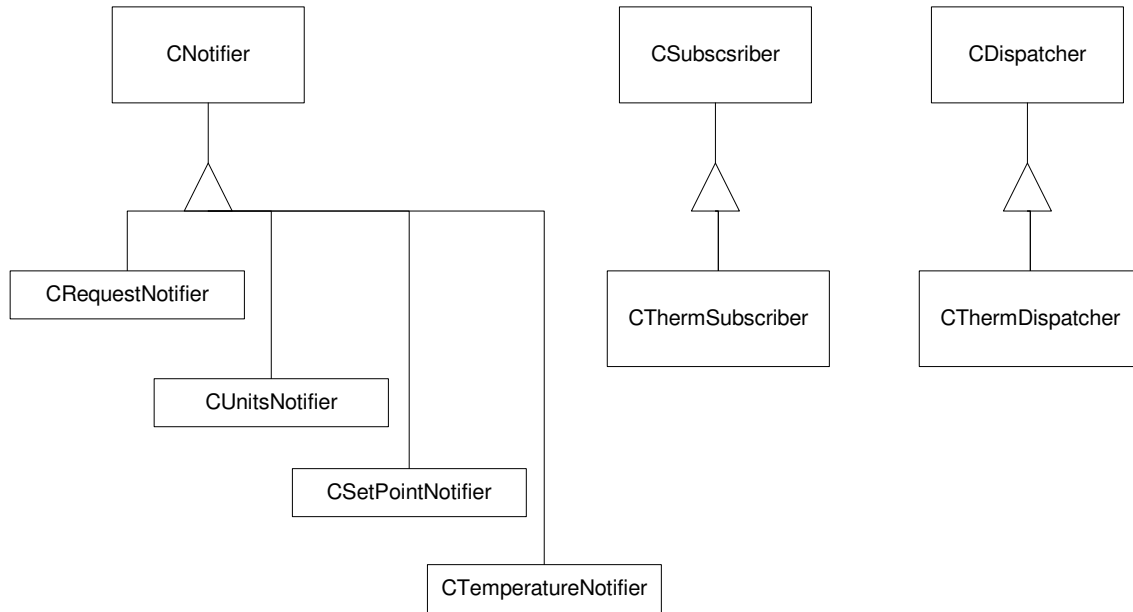
The dispatcher, maintains a list of all pending notifiers and all active subscribers. It receives a notifier from the publisher, sends the notifier to the appropriate subscribers, then destroys the notifier. Only the subscriber and notifier base classes are aware of the dispatcher.

The dispatcher runs in the application thread and dispatches notifiers at regular intervals triggered by a timer message.

¹ It's actually a virtual function in the application's subclass of CSubscriber, e.g. CThermSubscriber, as described below.



Notifier Class Hierarchy



CNotifier hierarchy

CNotifier is an abstract base class that is never directly instantiated. There's typically a different notifier class for each type of information sent between objects. For each notifier class, there's a virtual function in CThermSubscriber, and a case statement in CThermDispatcher::DispatchToOne.

For example: CSetPointNotifier contains the current setpoint value. In CThermSubscriber, there's an *OnSetPointNotifier* virtual function, and in DispatchToOne, there's a case statement that calls the *OnSetPointNotifier* function of a subscriber.

CSubscriber hierarchy

CSubscriber is an abstract base class that is never directly instantiated. Each application has a subclass for CSubscriber, shown above as CThermSubscriber. CAppSubscriber has a virtual function for each notifier subclass, for example, a CRequestNotifier is sent to a subscriber with a call to *OnRequestNotifier*.

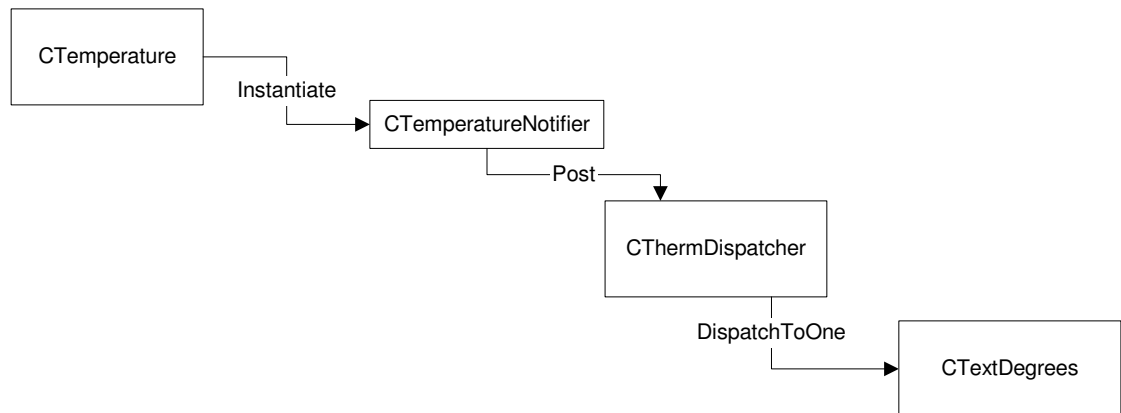
CDispatcher hierarchy

CDispatcher is an abstract base class that is never directly instantiated. Each application has a subclass for CDispatcher, shown above as CThermDispatcher, which is created during application initialization.

CThermDispatcher has only one purpose: to override the *DispatchToOne* virtual function. *DispatchToOne* is called by CDispatcher to dispatch one notifier to one subscriber. *DispatchToOne* is overridden to handle the notifier classes specific to the application.



The notification process



A single notification follows steps like this example:

1. The temperature sensor, which can be any object, calls *PostTemperatureNotifier*, which instantiates a *CTemperatureNotifier* and posts it to the dispatcher. This queues the notifier.
2. The dispatcher is subsequently awakened by its own timer, retrieves the temperature notifier from its queue, and calls the *DispatchToOne* function repeatedly; once for each subscriber that holds a subscription to the notifier.
3. *DispatchToOne* identifies the notifier as a `TEMPERATURE_NOTIFIER`, and in its case statement, calls the *OnTemperatureNotifier* function of the subscriber. Since *CTextDegrees* is a subclass of *CThermSubscriber*, and holds a subscription to temperature notifiers, its *OnTemperatureNotifier* function is called.



Getting Started

After you download and install the software, you'll probably want to build and test the sample program, to make sure it all works as advertised. Then you can follow the steps for integrating notifiers into your application.

Build environment

The error simulator is built with Visual C++ 5.0 using the MFC collection classes. If there's enough of a demand, I'll rewrite the notifiers using the STL collection classes, to remove the dependency on MFC.

Download and install

Extracting the files

1. Download the source files, if you haven't already. They are in Windows self-extracting file built with WinZip®.
2. Run the downloaded file and extract the notifier files.

Files and folders

Assuming you extracted into C:\ONTARGET, The files are organized as follows:

C:\ONTARGET\BIN\	contains binary files, including the notifier DLL and lib files, and the release and debug versions of the sample program. The projects are setup to write .exe, .lib, and .DLL files to the bin directory. To prevent name conflicts, all debug versions of DLL's are appended with the letter 'd'.
C:\ONTARGET\DOC\	contains this document
C:\ONTARGET\INCLUDE\	contains notifier header files that will be included in the sample program and in your own application
C:\ONTARGET\THERMOSTAT\	contains source code for the thermostat sample program
C:\ONTARGET\notify\	contains source code for the notifier library

Build the thermostat sample

To build and test the thermostat sample, you need the Microsoft Visual C++ development environment, version 5.0 or later.

Build the notifier DLL

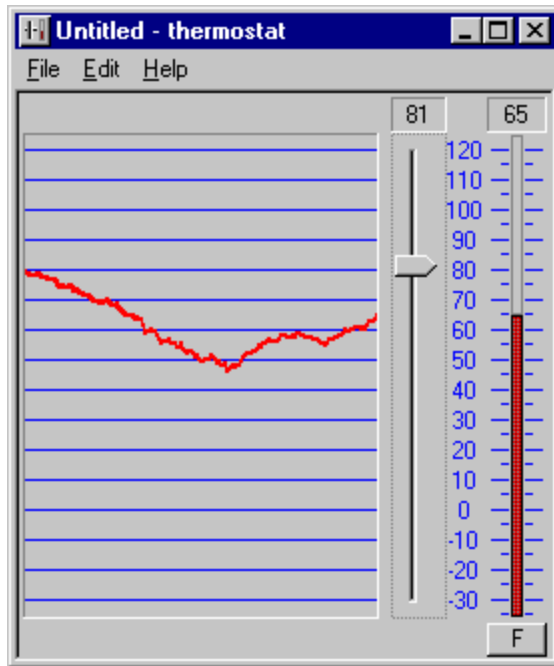
You could reasonably skip this step, since the `notify.dll` file is already present in the bin directory.

To build `notify.dll`, open the `notify.dsw` workspace in the `notify` directory. Build the release version of the notifier DLL. Unless you need to debug the notifier dispatcher, there's no need to build the debug version of the notifier DLL.



Build the sample thermostat program

Open the `thermostat.dsw` workspace in the `thermostat` directory. Build the debug version of the thermostat program. It's an MFC application that should look this when you start it up:



For a full description of how the thermostat example uses notifiers, please see the notifier article referenced in the overview.

Integrate notifiers into your application

1. Follow the four steps in the *Integration* section, below. When you're done, you'll be linking with the notifier library and you'll have all the customized subscriber and dispatcher classes you need to begin connecting objects.
2. Choose an item of information which is controlled by one object but which affects other objects when it changes. Encapsulate the information in a notifier, as shown in *Adding a Notifier Class*.
3. Create and post the notifier from the controlling object. This is described in *Using Notifiers: Publishing*.
4. For each object affected by the information, subclass `CSubscriber`, subscribe to the notifier, and override the `OnXXXNotifier` function. This is described in *Using Notifiers: Subscribing*.
5. Now, the subscribing objects are less dependent on the publisher; you may be able to remove direct pointer connections, if there are no other dependencies.
6. Repeat steps 2 through 5 for all loosely-coupled information. Continue to isolate objects, remove header and pointer dependencies as much as possible. Your application is becoming less complex and easier to maintain with each pointer and `#include` that you remove.

It's not a panacea

Don't go overboard. Notifiers make applications *more* complex when many large notifiers are sent frequently as a replacement for a direct pointer interface, or when an application has a hundred classes of notifiers. Use your judgment. Apply notifiers to simple information exchanged between loosely-coupled objects.



Integration

These are the steps to integrating notifiers into an application.

1. Create the NOTIFIER_CLASS enum
2. Subclass CSubscriber
3. Subclass CDispatcher
4. Create and initialize the dispatcher

1. Create the NOTIFIER_CLASS enum

Create an enum in one of your application header files, that lists the names of the notifier classes used in your application. The thermostat example uses this enum:

```
// Notifier Classes
enum NOTIFIER_CLASS
{
    REQUEST_NOTIFIER,          // sent to request an updated notifier
    TEMPERATURE_NOTIFIER,     // sent when the temperature changes
    UNITS_NOTIFIER,           // sent when the temperature units change
    SETPOINT_NOTIFIER         // sent when thermostat setpoint changes
};
```

You'll be adding to this enum as you add notifier classes.

2. Subclass CSubscriber

Each application needs its own subclass of CSubscriber. The subclass adds functions that respond to the application's notifiers. The subclass is typically implemented completely in the header file. See `thermSubscriber.h` for an example. The class definition is simply:

```
class CThermSubscriber : public CSubscriber
{
public:
    CThermSubscriber(SUBSCRIBER_PRIORITY nPriority = PRIORITY_NORMAL);
    CThermSubscriber(const CThermSubscriber &src);
    virtual ~CThermSubscriber(void);
    CThermSubscriber &operator=(const CThermSubscriber &src);

    virtual void OnRequestNotifier(CRequestNotifier *pNotifier)    {}
    virtual void OnTemperatureNotifier(CTemperatureNotifier *pNotifier) {}
    virtual void OnUnitsNotifier(CUnitsNotifier *pNotifier)      {}
    virtual void OnSetPointNotifier(CSetPointNotifier *pNotifier) {}
};
```

As shown above, there's an *OnXXXNotifier* function for each notifier defined in NOTIFIER_CLASS.



3. Subclass CDispatcher

Each application needs its own subclass of CDispatcher. The subclass overrides the *DispatchToOne* method, which calls the appropriate *OnXXXNotifier* function. Here's the method as defined for the thermostat example:

```
HRESULT CThermDispatcher::DispatchToOne(CNotifier *pNotifier,
                                       CSubscriber *pBaseSubscriber)
{
    CThermSubscriber *pSubscriber = (CThermSubscriber *) pBaseSubscriber;

    switch (pNotifier->GetClass())
    {
    case REQUEST_NOTIFIER:
        pSubscriber->OnRequestNotifier((CRequestNotifier*) pNotifier);
        break;

    case TEMPERATURE_NOTIFIER:
        pSubscriber->OnTemperatureNotifier((CTemperatureNotifier*) pNotifier);
        break;

    case UNITS_NOTIFIER:
        pSubscriber->OnUnitsNotifier((CUnitsNotifier*) pNotifier);
        break;

    case SETPOINT_NOTIFIER:
        pSubscriber->OnSetPointNotifier((CSetPointNotifier*) pNotifier);
        break;

    default:
        ASSERT(false);
    }

    return S_OK;
}
```

4. Create and initialize the dispatcher

When your application is initialized, create and initialize the dispatcher subclass defined in the previous step:

```
bool bOK = false;
m_pDispatcher = new CThermDispatcher();
if ( m_pDispatcher && SUCCEEDED(m_pDispatcher->Init()) )
{
    bOK = true;
}
```

When your application exits, destroy the dispatcher. This will also destroy any pending notifiers.

```
if (m_pDispatcher)
{
    delete m_pDispatcher;
    m_pDispatcher = 0;
}
```

Be sure to modify your link settings to link with `notify.lib`.



Adding a Notifier Class

a. Define the notifier subclass

A notifier class is typically defined completely in its header file using inline functions. This is because notifiers are lightweight, with only one or two data members.

Here's an example of the definition of the CUnitsNotifier class in the thermostat example, that signals a change from Fahrenheit to Celsius, or vice versa:

```
class CUnitsNotifier : public CNotifier
{
public:
    CUnitsNotifier(void);
    CUnitsNotifier(const CUnitsNotifier &src);
    virtual ~CUnitsNotifier(void);
    CUnitsNotifier &operator=(const CUnitsNotifier &notifier);

public:
    THERM_UNITS m_units;
};
```

In addition to defining the class and implementing the constructors, destructor, and assignment operator, the header file defines two inline functions that make it easy for publishers to post or send a change in the temperature units. See unitsNotifier.h for an example of this.

b. Add to the NOTIFIER_CLASS enum

The NOTIFIER_CLASS enum defines an ID for each notifier class.¹ If you add a notifier class, you need to add an entry for that class. See the code sample in Integration Step 1 above.

c. Add a virtual function to the CSubscriber class

In Integration Step 2, above, you created a CSubscriber subclass for your application. When you add a new notifier, you need to add a virtual function to the subclass that corresponds to that notifier. Subscribers that want to receive the notifier will override this function.

All you need to do is add one line to the class definition, like this:

```
virtual void OnUnitsNotifier(CUnitsNotifier *pNotifier) {}
```

d. Add a case statement to *DispatchToOne*

In your application's CDispatcher subclass, created in Integration Step 3, add a case statement for the notifier. This case statement calls the CSubscriber virtual function defined above:

```
case UNITS_NOTIFIER:
    pSubscriber->OnUnitsNotifier((CUnitsNotifier*) pNotifier);
    break;
```

¹ The dispatcher uses a bit array to indicate the notifiers a subscriber should receive. The notifier DLL has a value set at compile-time that determines that maximum number of notifier classes. In the DLL that I posted at my website, that number is 64. If you have more than 64 notifier classes, you can set MAX_NOTIFIER_TYPES in subscriber_mask.h to the number of notifier classes you have. The dispatcher needs a 32-bit word in the subscriber mask for each 32 notifier classes or any fraction thereof.



Using Notifiers

Publishing

Calling the Post or Send function

Any object can publish a notifier. You only need to:

- instantiate the notifier
- initialize its data members
- post or send the notifier

Most notifiers provide *PostXXX* and *SendXXX* convenience functions that do all this. Typically all the publisher has to do is call the convenience function, like this:

```
PostTemperatureNotifier(fTemperature);
```

Post vs. Send

Post appends the notifier to the dispatcher queue and returns immediately, before the notifier has been dispatched.

Send immediately dispatches the notifier to all its subscribers before returning to the publisher.

Subscribing

a. Subclass CSubscriber

Applications define their own *CSubscriber* subclass, as described in *Integration Step 2*. Objects inheriting from this subclass can receive notifiers. Their class definitions look like this:

```
class CTextDegrees : public CWnd, CThermSubscriber
```

b. Override the constructor

The object's constructor has to be overridden to explicitly call the *CSubscriber* constructor, like this:

```
CTextDegrees::CTextDegrees() : CWnd(), CThermSubscriber()
```

c. Subscribe to a notifier class

For efficiency, subscribers are only called for the subscriptions they request. To receive a notifier, a subscriber must subscribe to the notifier class. This is typically done at initialization:

```
HRESULT hr = Subscribe(UNITS_NOTIFIER);
```

d. Respond to the notifier

To respond to a notifier, override the *OnXXXNotifier* function, like this:

```
void CTextDegrees::OnUnitsNotifier(CUnitsNotifier *pNotifier)
{
    if (m_units != pNotifier->m_units)
    {
        m_units = pNotifier->m_units;
        DrawCurrentTemperature();
    }
}
```



CNotifier Class

Overview

CNotifier is a base class that is intended to be subclassed to represent a specific event. When the event occurs, the notifier is instantiated and either posted or sent.

Properties

m_nClass defines the notifier subclass. The base class constructor sets *m_nClass*. Each notifier subclass calls the base class constructor with its own class type, as shown below:

```
inline CTemperatureNotifier::CTemperatureNotifier(void)
    :CNotifier(TEMPERATURE_NOTIFIER)
{
    m_nTemperature = 0.0;
}
```

Applications typically define an enum to identify their notifier subclasses. The thermostat sample defines the NOTIFIER_CLASS enum.

m_nPriority defines the order in which notifiers are dispatched. Notifiers of the same priority will be dispatched in the order they are posted. Notifiers of different priority will be posted in priority order, where the highest priority has the lowest numerical value.

Methods

Constructor(void)

Use this to initialize all data to safe values.

Destructor(void)

Use this to free data objects and/or release COM objects.

Post(PRIORITY);

The notifier posts itself to the dispatcher. The dispatcher queues the notifier for subsequent dispatching. Returns zero on success.

The dispatcher deletes the notifier after completing the dispatch.

Send(AUTODELETE);

The notifier sends itself to the dispatcher. The dispatcher immediately sends the notifier. If another notifier is being dispatched, the sending of this notifier interrupts the current dispatch.

By default, the *autodelete* parameter is *true*, causing the notifier to be deleted after the dispatch. The publisher can set *autodelete* to *false*, leaving the notifier in existence. This allows subscribers to communicate with a publisher by modifying the notifier.



CSubscriber Class

Overview

The CSubscriber base class is a mixin class, which means it's intended to be subclassed using multiple-inheritance.

Further, an application must define its own CSubscriber base class, with virtual functions for each notifier class. To receive notifiers, an object must inherit from the application's CSubscriber subclass.

Properties

`m_nSubscriberPriority` defines the order in which a notifier is dispatched to the list of subscribers. Subscribers of the same priority are listed in the order they subscribed. Subscribers of different priority are listed in priority order, where the highest priority has the lowest numerical value.

Methods

Constructor (SUBSCRIBER_PRIORITY)

Sets the priority of the subscriber, but does not create a subscription for the subscriber (i.e. does not add the subscriber to the dispatcher's subscriber list).

Destructor (void)

Removes all subscriptions for the subscriber.

Subscribe (UINT nSubscription)

Add a subscription to a notifier class. Each application defines an enum, like NOTIFIER_CLASS, to identify the different kinds of notifiers.

Unsubscribe (UINT nSubscription)

Removes a subscription.

IsSubscribed (UINT nSubscription)

Returns *true* if the subscriber has a subscription to a notifier class.



CDispatcher Class

Overview

The CDispatcher class is a pure abstract base class that must be subclassed for each application. Specifically, the *DispatchToOne* method must be overridden. The dispatcher is then created and destroyed in the application object and otherwise hidden from all other parts of the application.

CDispatcher is a facade class that hides the interface of the CDispPrivate class from the subscribers and notifiers.

Properties

No visible properties – the dispatcher data structures are hidden in CDispPrivate.

Methods

Constructor (void)

Initializes dispatcher to safe values

Destructor (void)

Deletes the private dispatcher object.

Init (UINT nTimerPeriod)

Creates the timer that triggers periodic dispatching. The timer period is specified in milliseconds.

IsSubscribed (CSubscriber *pSubscriber, UINT nSubscription)

Return *true* if a subscriber has a subscription to a given notifier class.

Post (CNotifier*)

Add a notifier to the dispatch queue.

Send (CNotifier*)

Immediately dispatch a notifier to all its subscribers.

SetSubscriberPriority (CSubscriber *pSubscriber, SUBSCRIBER_PRIORITY nPriority)

Assign a new priority to a subscriber.

Subscribe(CSubscriber *pSubscriber, UINT nSubscription)

Add a subscription to a notifier class for a given subscriber.

Unsubscribe(CSubscriber *pSubscriber, UINT nSubscription)

Remove a subscription to a notifier class for a given subscriber.

UnsubscribeAll(CSubscriber *pSubscriber)

Remove all subscriptions for a given subscriber.



More Free Software

Error Simulation

Error simulation is a technique for thoroughly testing a program's error handlers by inserting C++ statements that force error conditions.

What is an error handler?

Anywhere a program tests the return code of a function or catches an exception, it's handling an error. In my own software, I've found that testing, propagating, and responding to errors makes up between 7% and 34% of the executable lines of code in my own programs, averaging about 20%. It's lowest in modules that have few external interfaces. It's highest in I/O modules and routines that interact heavily with the system.

Why test error handlers at all?

Error handlers don't normally execute so they get less attention than feature-driven software. Instead, they're entirely responsible for robustness: the ability of a program to continue in the face of hardware failures, system errors and resource limitations. Robustness is important to commercial applications, but it's essential to embedded and mission-critical software that cannot afford to abort or corrupt their data when handling failures.

Only the most trivial software has the luxury of aborting without cleaning up. Most programs will need to close files and databases, or release communications ports or shared memory. Without testing error handlers, you can safely assume the cleanup will fail.

What is error simulation?

Error simulation is a developer's tool for finding defects in error handlers by forcing errors at the point of detection. Immediately after calling a function that can fail, the programmer inserts a call to a function that simulates the failure, either by returning a failure code or by throwing an exception.

Why test with error simulation?

You can perform limited tests of error handlers by running on a machine with low memory or faulty hardware. This is hit-or-miss and will execute only a few error handlers. Worse, tests are nearly impossible to reproduce, making it difficult to fix problems. Regression testing is hopeless.

The error simulator exercises all error handlers in a repeatable way that can be automated with commercial test software.

For many products, testing and fixing all error handlers using error simulation is faster and cheaper than finding and fixing a single bug in a single error handler, when the only facility for reproducing the bug is an intermittent hardware failure.

Debug message logging

Coming soon to the On Target Software website – the humble *printf* with a new suit of clothes. Too many times I've found myself working on problems where a debugger is useless. Interactive software that depends on change-of-focus can't be fixed with a debugger. Problems in the release configuration can't be touched with a debugger. For those times, and as a supplement when working on tough bugs, I need to trace program functions.

For this reason, I've developed a set of debug functions that work like *printf* but write to a circular log file. They are conditionally compiled, of course, so you can remove them from the release version. But you can also leave them in the release version – for debugging, on-site logging, etc.



Why all this free software?

I'm a software contractor and my products are the prototypes, designs, and software that contribute to my clients' commercial applications. The work that would best advertise my skills is the confidential property of my clients. I wouldn't have it any other way, but that means I have no visible portfolio.

So I decided to take some of my ideas, those few that are general purpose, neither confidential nor proprietary, and deliver them as software components or utilities through my website.

I hope you find the software useful. If you think it is and you think I could make a more direct contribution to your C++/Windows project, please call me at (781) 834-6542. I work with companies throughout North America.

*Thanks,
Dave*

How to reach On Target Software



ON TARGET
software

dave.pomerantz

56 bartlett's island way
marshfield, ma 02050

781/834 6542 phone

781/834 6416 fax

<http://www.targetsoft.com>

dave@targetsoft.com

Revision History

1.0 Tuesday, November 25, 1997

First version of document.

1.1 Sunday, July 19, 1998

Fixed bugs in requestNotifier.h, setpointNotifier.h, temperatureNotifier.h, and unitsNotifier.h. These functions would return E_OUTOFMEMORY from Send and Post functions, even when the allocation succeeded. The defect wasn't discovered during testing because the callers, which are all void message functions, can't propagate errors so they don't check return codes. To detect this bug, I should have used the error simulation to check return codes even when return codes aren't used.

Thanks to Richard Schuller for finding and reporting the problem.

1.2 Thursday, November 6, 2008

Still a lot of people using C++, so there may yet be a use for the notifiers. I rebuilt the code and fixed compiler errors (mostly related to the new security-enabled versions of Microsoft string functions) and updated the text of this document.