DevPlus

Developers Guide

Table of Contents

Legal Information

This document is provided as part of the DevPlus product which is released under GPL licence. As a recipient of the GPL product you are granted rights to distribute this document along with DevPlus as long as you follow the requirements of the GPL licence.

However, this document itself is not released as GPL. You are not given permission to modify this document or produce derivative works of it and distribute without permission by the author.

You are of course, will within your rights to create your own guides, manuals, or documention for the DevPlus library without needing any special permission from the DevPlus authors.

Introduction

DevPlus is a bunch of classes that maintain a sensible interface as closely as possible throughout the various classes and functions. It is designed to be a powerful substitute or enhancement to the various incompatible methods within MFC and other libraries.

DevPlus is intended to be compiled into the application, rather than linked in. This has some added advantages. Of course, if you wanted you could create a library out of it and link it in however you want.

DevPlus provides functionality for Thread, Socket and Text handling, as well as detailed support for database access (MySQL and generic ODBC).

Any functionality that can be re-usable for different projects will likely end up in DevPlus over time. When possible it is also intended to be compilable on multiple platforms.

Document History

Feb 25, 2005
Started writing the document based on the current version of 0.1.36

Version Compatibility.

DevPlus is constantly being updated with new features, and bug fixes. To ensure that applications and developers are not adversely affected by changes in new versions we have provided a comprehensive versioning system with a good chunk of historical information stored in the actual library.

This means, that if you start working on an application using version 0.1.27 which has a particular set of objects and function calls, but a new version comes out (0.1.36) that both adds features, removes functionality that is no longer needed, and fixes some bugs throughout the code. The developer has the choice in updating the application to use the new features, or set the VERSION_DEVPLUS macro to 127. By setting the macro to a previous version, the interface and behavious will mimic the 0.1.27 version, but all bug fixes that affect that code should be active. So you have some bugs fixed, but you dont have to change your application code.

That being said, it is recommended that developers use the features of the latest version that is available, as there are points in time where much older versions of DevPlus are no longer supported within the code.

Compatibility

DevPlus was originally developed for DOS, ported to Windows, and then finally ported to Linux. Most DevPlus development is currently being done for Linux, however it should work just fine with windows also as we attempt to make it easily cross-compilable. The purpose of this library however is to create an interface that works on multiple platforms without requiring any modifications to the source code.

Most examples in this guide will be for Linux, and if there is ever a need it can be adjusted for Windows.

DevPlus is also packaged for ArchLinux for easy integration. However, keep in mind that the pre-build object file is compiled as the very latest version with Threads turned on. If you wish to compile a different version of DevPlus you will need to do that yourself as part of your application.

Using DevPlus in your Application

If you are using ArchLinux, you can simply install the package, and then in your applications include <DevPlus.h> when needed, and then add the -ldevplus-thread linker command. And since most DevPlus operations are thread-based, you would also want to link in the pthread library.

For example:
g++ -o main main.c -ldevplus-thread -lpthread

If you wish to compile with your own options, then it is recommended that you create a symbolic link in your application source folder to point to the header and source file for DevPlus which would normally be in:

/usr/src/DevPlus/DevPlus.h
/usr/src/DevPlus/DevPlus.cpp

Compiling single thread applications

By default, DevPlus uses threads. However, in some circumstances developers need to ensure that an application is single threaded. To ensure that all threading code is not compiled, set the macro DP_SINGLETHREAD and all objects that require threading will be excluded.

Excluding some functionality

Some developers may want to exclude particular functionality from being compiled. DevPlus will ensure that any objects which require functionality that is excluded will also get excluded. Ensuring single-thread functionality is done this way, by excluding the basic objects that all threadable objects require.

To exclude functionality, you set a specific macro value in your makefile.

_EXCLUDE_SOCKET
_EXCLUDE_THREADBASE
_EXCLUDE_READWRITELOCK
_EXCLUDE_THREADOBJECT
_EXCLUDE_SERVERINTERFACE
_EXCLUDE_DB
_EXCLUDE_DB_ODBC
_EXCLUDE_DB_MYSQL

Here is an example Makefile:

```
all: sample

OBJS=sample.o DevPlus.o
LIBS=-lpthread
DPOPTIONS=-D_EXCLUDE_SOCKET -D_EXCLUDE_DB

sample: $(OBJS)
     g++ -o sample $(OBJS) $(LIBS)

sample.o: sample.cpp sample.h DevPlus.h
     g++ -c -o sample.o sample.cpp $(DPOPTIONS)

DevPlus.o: DevPlus.cpp
     g++ -c -o DevPlus.o DevPlus.cpp $(DPOPTIONS)

clean:
     @-rm $(OBJS)
```

This Makefile ensures that the same options are used for each object file that is compiled. This is important because some data structures may be different depending on the options used and can cause applications to crash in weird ways.

Scope Limitations

DevPlus does not attempt to do everything possible. Instead it is a group of common objects used by the author in various projects. There is a large amount of functionality that has not been incorporated because existing libraries exist that should be used instead. However, the author noticed that there is no cross-platform threading object that provides the features that we provide, and those features were required for something. Also, the author wanted a simple method to listen on a port for incoming sockets, and an object that can easily communicate over that socket.

ASSERT's and catching errors

DevPlus provides and uses an ASSERT() macro which is used to catch potential programming errors. It is intended to assist in catching implementation errors by developers who dont quite understand the procedures of the library.

For example, with the DpSocket::Receive function, the first parameter is a char pointer. The developer may missunderstand and think that the function will create some memory and return that. Which is not how the function works, so if the developer calls that function with a NULL, or a len of zero, then an ASSERT is generated because obviously the developer did not understand how this function is used. Alternatively, a developer might use some sort of algorithm to determine the parameters that will be supplied, but a bug in the algorithm causes a length of 0, or a NULL pointer to be provided. If DevPlus ignored those errors and just continued on, the developer may not ever realise that an error in the algorithm exists, and testing may never produce the problem.

Mistakes can be made, and if those mistakes can be detected, or rather, if we can attempt to ensure the integrity of the information we have, then simple bugs can be found. On Linux, the application will stop when an ASSERT fails. On windows, under VisualStudio, an ASSERT will pause the application and generate a message asking the user if they want to debug it, stop, or continue.

ASSERT's are also put in the DevPlus code as often as possible to catch any bugs that we accidentally introduce. For example, at one time a bug was fixed in DevPlus, but a typo caused the function to exit pre-maturely. An ASSERT caught this bug because the data was in a state that wasn't exactly accurate. However, the bug might not have ever been noticed, but could have definitely caused data corruption.

Developers are encouraged to use ASSERT macro's in their own code to try and ensure that bugs or typing mistakes have a chance of being detected early.

NOTE: Do **NOT** use ASSERT's for general condition testing. Because on different systems ASSERT's are treated differently. You should program your applications so that an ASSERT is never raised unless a bug exists. On Linux, a ASSERT will print some information about it, and then terminate the application. On Windows (when compiled in DEBUG mode) an ASSERT will pause the application and show a dialog box asking the user if they want to ignore, debug, or terminate. End-users should never see this. Additionally, you should never assume that the application will actually terminate when an ASSERT is fired.

Object Reference

This section details all the classes, a description of how it works, and a detailed listing of the methods and elements within the object.

DpThreadBase

This object is used by any other object that either needs to be thread-safe, or will interact with a thread, or become a thread. It is rather internal to DevPlus and is not really intended to be used by application code.

You can disable this object (and also any that require this one) by setting a global macro such as:

```
#define _EXCLUDE_THREADOBJECT
```

Alternatively, to ensure that your application does not have any threaded code compiled in:

```
#define DP_SINGLETHREAD
```

`void Sleep(DWORD dTime)`

This function will Sleep for *dTime* miliseconds. In windows, you get the ability to sleep for miliseconds, but linux allows for micro-seconds, and different functions are required. This function provides a consistant interface for the different platforms.

DpThreadObject

This object would allow a program to derive a class and it would become a thread-running class. The derived class would require several virtual functions to be used. These would be used to Start and Stop the thread, as well as a function that will be Run every cycle.

This object could also be used by itself so that threads can be started and controlled with a simple interface.

For example, this class can be used in the following example:

```
class CSample1 : public CThreadObject
{
    public:
        CSample1() { Start(); }
        virtual bool OnThreadRun(void);
};
```

This example would basically make each instantiated object a thread, and will execute the ThreadRun function constantly.

The constructor in this example starts the thread as threads dont start automatically, your derived class must decide when the thread should be started. Since this example, does not make a call to SetCycleTime it is assuming that the over-ridden OnThreadRun() function will run one time, and then the thread will exit.

`void SetCycleTime(DWORD dTime=DP_DEFAULT_CYCLE_TIME)`

The thread object can be put in an alternative mode which will cause the OnThreadRun() function to be called periodically. This is useful for objects that want to perform a particular function every 250 miliseconds for example.

This function should be called before the thread is started. Once started, if this function is called, it is likely to cause an ASSERT and it would be too late for it to take affect if the ASSERT is ignored.

`void SetStackSize(unsigned nStackSize)`

In windows, each thread is given its own stack which is of a default amount. When the thread is started, you can specify how much stack space to allocation. If you know that your thread will not be making many recursive calls and is safe to use a smaller stack space, you can lower memory usage by providing a smaller value than that default. Alternatively if you know your object will need a lot of stack space, you can increase this amount.

This function must be called before the thread is started.

This will compile for all platforms but is only affects windows compilations.

void Start(void)

Assuming that the derived class has an OnThreadRun virtual function, Start() will activate the thread. This function is *protected* so it can only be run from within the derived class itself.

void Start(void (*fn)(void *), void *pParam)

DpThreadObject can be used without creating a child-class. You can start the thread and pass it a call-back function that will be given one void pointer parameter.

void Stop(void)

The thread can be stopped, assuming that it is in cycle mode.

virtual void OnThreadStart(void)

This virtual function is called before a thread is completely started. Especially useful when the thread is used in cycle mode, but not limited to that mode. The function is actually performed inside the thread, and is basically the first thing that is performed.

virtual void OnThreadStop(void)

Over-ridden function that is executed when a thread is stopping. This is called after the Stop() command is received, and after OnThreadRun completed if it was running at the time. It is executed within the thread, just before it is stopped completely.

virtual void OnThreadRun(void)

This virtual function is called once when not in cycle mode, and called every dTime miliseconds when in cycle mode. Cycle's are attempted to be calibrated, which means that if you want the function to execute every 1 second (1000 miliseconds), it will take into account how long it processes for before sleeping, so that it should activate 1000 miliseconds after it was previously activated.

virtual void Lock(const bool write)

Used to safely lock the thread contents so that multiple threads, generally the running thread and the calling thread can modify values without stepping on each others feet. *write* indicates whether it is a read lock or a write lock.

virtual void Unlock(const bool write)

Unlock. Opposite of Lock(), and MUST be called when finished using a Lock().

DpReadWriteLock

This class can be used to implement read/write locks. If you want to lock an object for reading where multiple threads can read from the object. When a thread needs to modify the object, then it will do a write lock which will block any new read lock requests and wait until all of the current read locks have been removed, and then it will actually lock the object. When the write lock is removed, the pending read or write locks will start contending for the object once more.

Note: It is important to keep in mind that if a particular thread locks an object for reading and then locks it for writing, that first read will never be removed (because the thread is waiting for the write to succeed), and the system will be dead-locked. An assertion in some cases will detect this and provide a warning but developers cannot rely on this alone and need to be ultra careful when setting a read lock. If in doubt, always set a write-lock.

`void ReadLock(void)`

`void WriteLock(void)`

`void Unlock(void)`

DpSocket

This is a socket functionality wrapper. Basic functionality for connecting, sending and receiving data over a socket. Not all socket features are implemented. Interface is simple and used from a functional perspective rather than a direct mapping to the api. Basically, you connect, send, receive and close.

void Init()

SOCKET Accept()

void Accept(SOCKET)

SOCKET Detach()

bool Connect(char *szHost, int nPort)

int Send(char *data, int len)

int Receive(char *data, int len)

void Close()

bool IsConnected()

SetNonBlocking()

DpDataQueue

This is a simple implementation of a FIFO queue. It uses single-char elements and allows you to add, view and remove data from queue. To edit data in the queue, you need to get the data, copy it somewhere, edit it, clear the queue ad then add the data back.

Because of the way this class is buitl, it can be used for both stack and FIFO (first in, first out) queue. So if you are using it as a FIFO queue, make sure you use Add/Pop functions. For a stack, use Push/Pop. Operations can be used interminably. That means that if you pop data off, and want it put

Links

ArchLinux
http://www.archlinux.org/

Index