[This document was manually converted from the original M4KIT.HTM document in HTML format]

Introducing the MetaKit library

Thank you for your interest in the MetaKit library. This document describes the main features and requirements of this software product, shows you how it can be used with some coding examples, and tells you how to find your way in the different classes and files which are present in this package. There is a separate section which describes some sample applications based on MetaKit, and for which complete source code is included.

A class library to manage structured data?

There are many ways to develop software, there are many languages, and nowadays, there are as many object oriented class libraries. Correction, that was yesterday. Today, these are called frameworks. Well... here's an unusual one. In C++.

The MetaKit class library does only one thing. It takes care of the data you give it: manages it, stores it on file, serializes the data for stream I/O, and it loads it back in on request. Persistent data, data manipulation, that's the general idea here. It can instantly restructure data files, allowing you to adjust your design as often as you like.

Are you looking for an SQL database interface? *Sorry, wrong number.* Do you wish to store data in DBF or WKS format? *Sorry again.* Would you like to store everything with ODBC, BDE, JET? *Nope, this library is not what you want.*

- Do you want to create self-contained applications which need to store all sorts of data?
- Are those INI files and cleverly designed ASCII files starting to become far too complex?
- Do you hate to have to constantly go through file format conversions?
- Are you getting tired of converting your internal data structures to save them on file?
- Do those serializable objects seem to take forever to load on startup?
- Are you worried about losing data if the system fails at the wrong time (it will, eventually)?
- Do you wonder why the database you're using has tripled the size of your executable file?
- Are you worried about the support needed for all those DLLs or database drivers?
- Do you want to create document-centric software, using a single file to hold all data?
- Are you lost in all those header files and documentation sets that came with your database?
- Do you want a small database package? With a small price tag? And source code?

Yes? Welcome! Please step right this way... ***meet the MetaKit library!***

The latest news can always be found at http://purl.net/meta4/metakit on Internet.

REQUIREMENTS

The MetaKit library is a tool in the form of a high-level software (class) library. If it does what you expect, if it conforms to your development choices, if it's easy to work with, and if it can be incorporated in your final product, this might be just what you need. You are the only person who can answer the first question, but here are some notes on the other three issues.

C++ development

This release is intended to be used in combination with the Microsoft Foundation Class library (MFC). Strings, files, and several types of containers from this application framework are used by the MetaKit library. This avoids duplication of effort and code and leverages the performance and quality of this commercial-grade class library.

At this stage, the library has been built and used with Microsoft Visual C++ versions 1.52 (for both Win16 and MS-DOS executables) and 4.0 (for Win32 executables), as well as with Symantec C++ 7.2 (Win32) and Watcom 10.5 (Win32). There is no reason why another development environment using MFC should not work, but you will have to try it to find out.

This software has been designed with portability in mind, all current MFC and Windows dependencies are going to be removed in future versions. But for now, the focus will be on functionality and completeness.

The learning curve

Like any other C++ class library, the MetaKit library is not for the novice C++ programmer. It takes some effort to understand the intended use of classes and objects and to become familiar with a certain way of doing things the "nice" way. It takes determination to dig into the class details when things are not working quite as you expected. Failure to invest time to understand how all classes interact and where specific functionality is meant to be added or altered can lead to disappointing results. You should plan on joining forces with the class designers, not fight them. You even have to be willing to synchronize your programming style with others to benefit from their work. It is the class designer's task to reduce this effort as much as possible, but there is no way around understanding each other's needs and intentions.

Having said that, you may be surprised by the simplicity of MetaKit's API. The main header files are small. There is no deeply nested class hierarchy you need to study and understand. There is no base class that you need to be aware of for your own objects, and you will only rarely derive from classes in this package. In practice, most objects in MetaKit are either used as is or added to your own classes as member objects. This does not imply that MetaKit is a trivial piece of software (it isn't), but merely that a lot of effort has gone into the encapsulation of the underlying complexity.

As a consequence, the MetaKit classes tend to fit in nicely with a range of other class libraries (user interfaces, data communication, networking, even other database packages). You can start using this library for evaluation in your existing projects without disrupting existing structures and then choose to adopt more of it if it suits your needs. The design is highly modular (and will remain so as much as possible), allowing you to take what you like, without pulling in code which you do not wish or need to use.

Fair play

The shareware distribution method chosen for MetaKit is an experiment... you get a completely functional product for free, and you are expected to pay for this product if you intend to use it longer than the 30-day evaluation period. The product itself is anything but an experiment: the concepts have evolved and have been refined in commercial projects and products over a period of several years.

No, there are no artificial restrictions in this shareware product. Yes, the author would like to earn his money with this package and to continue further development. By not paying the fee associated with this product - which is absolutely minimal by any standard - you are in effect saying "Don't bother, I don't want this software or anything that could come out of it". Lack of income from this endeavour will cause the author to go back full-time to his fee-based consultancy and software development of niche products with 4 and 5-digit price tags. Sure, you can have a free lunch now, but then you'll never find out what's for dinner... You decide.

CODING EXAMPLES

The MetaKit library manages structured information for you, but it does this using a new approach. The best way to introduce many of the ideas is with sample code (if you're not a C++ programmer: shutdown now, reboot, and go have a nice cup of coffee).

Basics

Let's create a trivial address book with a single entry in it:

```
c4_StringProp pName ("name");
c4_StringProp pCountry ("country");
c4_View vAddress;
c4_Row row;
pName (row) = "John Williams";
pCountry (row) = "UK";
vAddress.Add(row);
```

In this code, we define two fields (called "properties"), a collection (called a "view"), and a temporary buffer (called a "row"), and then we set the properties (fields) to some sample values and add the row (entry) to our view (collection). Note that a property must be given a name when defined.
The most striking difference with the C++ code you've probably grown used to, is that the data structure has been defined entirely in terms of existing classes. What MetaKit does is to introduce "run-time data structures", i.e. structures which can be set up and manipulated entirely at run time. Big deal? Well... you're right, this doesn't look like much. In fact, the more trivial it looks, the better: using run-time data structures in your C++ sources should be (almost) as easy as the classical structure/class & member approach. Let's add a second entry:

```
pName (row) = "Paco Pena";
pCountry (row) = "Spain";
vAddress.Add(row);
```

Operators

Now let's add a third one using some shorthand operators:

```
vAddress.Add(pName ["Julien Coco"] + pCountry ["Netherlands"]);
```

Even if this looks a bit confusing at first, don't worry. There are only a couple of overloaded operators, and this is as concise as it gets. MetaKit is not a wild collection of non-intuitive operator notations, although the use of the array operator in the example above is definitely a bit unconventional.

The five operator notations you need to be aware of in MetaKit are:

view [index] - *is a* - ***row*** (this makes views similar to ordinary arrays)
property (row) - *is a* - ***value*** (which can be used on either side of an assignment statement)
property [value] - *is a* - ***row*** (a single row with a single property, sort of a constant, or "scalar")
row + row - *is a* - **row** (the concatenation of the property values in both rows)
property, property, ..., property - *is a* - ***view*** (an empty one with the specified properties)

Now that we've appended three entries, let's take some information out of the view again:

```
CString s1 = pName (vAddress[1]);
CString s2 = pCountry (vAddress[1]);
printf("The country of %s is: %s\n",
         (const char*) s1, (const char*) s2);
```

Persistence

Ok, but what about storing this information on file? Ah, well, that's real easy with these run-time data structures:

```
#define FORMAT "address[name:S,country:S]"
c4_Storage storage ("myfile.dat", FORMAT);
```

```
storage.View("address") = vAddress;
storage.Commit();
```

You have to describe the format of the data file. This is a string of the form "table1[prop:type,...],table2[...]". This format is simple but pretty strict - do not add spaces or mix up capitalization. The possible types are currently S (string), I (integer), and F (float).

Loading this data from file again is even simpler. You only need to make sure that properties are named and typed according to the stored structural information. Here is the full code to access a previously stored datafile:

```
c4_StringProp pName ("name"), pCountry ("country");
c4_Storage storage ("myfile.dat");
c4_View vAddress = storage.View("address");
for (int i = 0; i < vAddress.GetSize(); ++i)
    ... pName (vAddress [i]) ... pCountry (vAddress [i]) ...
```

There is no need for a format specification, the c4_Storage constructor used here opens an existing data file for reading and reloads the stored format.

You can have on-demand loading: data will only be read from file when actually needed. To do this, the storage object must not be destroyed: in MFC, making the storage object a member of your derived document class will do the trick. If a storage object is destroyed while there are still views referring to its contents, all relevant data will be loaded into memory. This may require a lot of time and memory space - so be sure to destroy or clear all views if this is not what you want.

Sorting

Now for some data manipulation. Let's create a sorted version of this view:

```
c4_View vSorted = vAddress.SortOn(pName);
```

The result is a new derived view with all rows sorted by name. Note that vSorted is not a copy, but that the rows in this view share their contents with vAddress. This is only another way to look at this information. Using a nasty overload of the comma operator, we can define composite keys, and thus sort on more than one property (the extra parentheses in the following line are essential):

```
vSorted = vAddress.SortOn((pCountry, pName));
ASSERT(vSorted.GetSize() == vAddress.GetSize());
```

There are several more functions (descending sorts, searches, selection on values and ranges), which will all work with any structure you care to build using the basic mechanisms just described. These manipulation functions can hide quite a bit of complexity. Here is a more advanced example:

```
c4_View vSome = vAddress.Select(pCountry ["UK"]).SortOn(pName);
for (int i = 0; i < vSome.GetSize(); ++i)
{
    CString name = pName (vSome[i]);
    printf("'%s' lives in the UK\n", (const char*) name);
    printf("Entry 'vAddress[%d]' is that same person\n",
                        vAddress.GetIndexOf(vSome[i]));
}
```

This will show a list of record indexes of all people living in the UK, sorted by name. The effect of the GetIndexOf() member is to "unmap" all selections and sorts back to the underlying view.

That's it. This is how MetaKit is used. Isn't it great? End of story.

Wait... is that all?

Not quite. There is more. One of the nice features of MetaKit is, that the views you just saw can also be stored inside other views. A view is very similar in a way to the integers and strings used up to this point. Let's first create a list of John's telephone numbers:

```
c4_StringProp pType ("type"), pNum ("num");
c4_View view;
view.Add(pType ["work"] + pNum ["+44 (1) 123 4567"]);
view.Add(pType ["home"] + pNum ["+44 (1) 123 6789"]);
```

Now, we can add this list to the original view by defining an appropriately typed property:

```
c4_ViewProp pPhone ("phone");
pPhone (vAddress[0]) = view;
```

This creates a nested view structure (or "repeating field", if you like). Our simplistic address book now holds three addresses, of which the first contains a list of two telephone numbers. Views can be as complex as you like, and can accommodate a wide range of application storage structures.

Storing such a compound data structure is simple... you can use almost the same code as before:

```
#define FORMAT "address[name:S,country:S,phone[type:S,num:S]]"
c4_Storage storage ("myfile.dat", FORMAT);
storage.View("address") = vAddress;
storage.Commit();
```

The only difference is the definition of the data format, which now includes the phone number list as a nested sub-view. Without this adjustment, everything except the new phone number lists would be stored on file as before. Existing data will be instantly restructured to match the specified format, but these changes will only affect the data stored on disk if and when they are actually committed.

In practical use, you will create a number of properties with appropriate names and datatypes and make them available via a header file. Although not required, it is far simpler if these properties are defined as global objects and exist as long as your application runs. Note that properties (objects of class c4_Property or classes derived from it) only act as (lightweight) placeholders, and that they are independent of actual data stored in views (in the same way that member fields are defined as part of a class in C++, long before objects of that class are created).

Performance

So far, nothing has been said about performance. For a reason: there is both good and bad news. The bad news is that this release has not yet been optimized for efficiency. Hundreds of entries will be fine, thousands are usually fine, but tens or hundreds of thousands... you may not like it. With on-demand loading, opening a file is instant regardless of file size, but the performance degradation becomes apparent when you start accessing or altering information.

Then again, the CatFish disk catalog browser utility built with the current version of MetaKit demonstrates that - even with so little optimization - excellent performance can be achieved if the data structure is designed to take advantage of the unique way in which MetaKit manages its data.

But the best news is, that performance is likely to increase dramatically in the next releases. This is based on results obtained with an early version of MetaKit which has shown stunning performance (on millions of objects) in a commercial application, but which used a less general class library interface. The current product is a major rewrite of that software to build a more general foundation, with emphasis on functionality first, speed second.

CLASSES AND FILES

This section presents a general perspective on the MetaKit classes, header files, and library files.

Class hierarchy

The outside view of the class hierarchy is almost flat: the public classes don't share a common base class, nor do they require complex derived classes or large numbers of virtual functions. As you might expect, there is essentially one class for every type of object:

c4_View For views, which can hold zero or more rows of data.
c4_Row For rows, consisting of zero or more properties, each with an associated value.
c4_Property The base class for properties, with a few basic derived classes.
c4_Cursor This is a generic iterator for views (or a "pointer to a row" if you prefer).
c4_RowRef A reference to a row, either a c4_Row, or one of the entries of a c4_View.
c4_Storage Objects of this type manage persistence and on-demand loading.

That's basically it. If you study the definition of the above six classes you will be ready to use MetaKit. The c4_View class might look quite familiar: it has the same set of member functions as the "C...Array" classes in MFC. The c4_Cursor class has all the operator overloads (++, --, *) you would expect for iterators.

Naming conventions

All class names start with the prefix "c4_". Until C++ namespaces become more widely implemented, this class library needs to use yet another naming scheme which must not conflict with whatever other software and class libraries you may be using. The decision was made to insert the digit four in every globally accessible identifier of this class library. In addition, the first letter of all identifiers indicates its type. It may not win a prize in esthetics, but it seems unlikely that anyone else uses this convention.

The most common prefixes are:

c4_ Classes
d4_ Preprocessor defines
f4_ Global functions
t4_ Typedefs

Two other conventions are used throughout this class library: function arguments ("formal parameters") always have a trailing underscore, and private member fields always start with a leading underscore. These choices do not affect your own programming style, but it helps to be aware of them while going through the headers and sources.

Header files

There are three categories of header files:

m4kit.h The top level header file you need to include in your sources.
k4*.h Contains all public class definitions of MetaKit.
k4*.inl Inline definitions included by the "k4*.h" files.

The two central header files of the MetaKit library are "k4view.h" where all the core classes are defined, and "k4conf.h" (a simplified version of the headers used to build te library itself) with the definitions to deal with hardware / operating system / compiler / framework dependencies (for now these are: x86 /

Win+Dos / MSVC / MFC).

As you can see, all public files contain the digit four to avoid name conflicts with any other files you may be using.

Dynamic libraries

The current version of this library is well suited for use as a Windows DLL. The library can be built in a range of configurations, but given MFC's concept of an "extension DLL" to supplement MFC itself, this is definitely the preferred interface for the library.

The DLLs come in two flavors, one for 32-bit and one for 16-bit Windows executables:

mk4nvmx.dll The dynamic link library for 32-bit Windows (requires mfc40.dll and msvcrt40.dll).
mk4nvmx.lib The import library to link with in MSVC 4.0.
mk4wvmx.dll The dynamic link library for 16-bit Windows (requires and extends mfc250.dll).
mk4wvmx.lib The import library to link with in MSVC 1.52.

Debug versions of these libraries (which are several times larger than the above files) are available to all registered developers. These add numerous assertion checks sprinkled throughout the code which will help with debugging by alerting you as soon as any inconsistency or incorrect use of the library code is detected.

Static libraries

MS-DOS does not support DLLs. To build stand-alone EXE files, you can use a statically linked version of the library:

mk4dvms.lib This is the large-model / real-mode / 16-bit version of the library.

You currently also need to link with MFC's "lafxcr.lib" to prevent linker errors for several of the classes in MFC (such as CString, CFile, and C*Array). Since no user interface code is used, the extra code added due to MFC is quite limited.

Static libraries for both 32-bit and 16-bit Windows are available to all registered developers, as are DLL versions which do not require any other DLLs to run (they were called "User DLLs" in MFC 2.5). The statically linked versions allow you to produce stand-alone applications which do not require any additional DLLs and are therefore easier to deploy and distribute.

SAMPLE APPLICATIONS

These sample applications illustrate a range of aspects of the MetaKit library in actual use.

DISCAT.EXE

This application is a good starting point to learn about MetaKit.

What it does: DisCat is a very simple disk catalog application. It recursively scans a directory tree on your hard disk and creates a data file with information about all files and subdirectories it finds during that scan. The total number of directories scanned is reported on the screen.

What it illustrates: This sample program shows how to set up a simple MFC-based application which uses MetaKit. The source code of this application can be compiled for 32-bit Windows as well as for 16-bit Windows targets. DisCat has no practical use, other than to generate a not-quite-trivial datafile using the MetaKit lib

How to use it: When you launch DISCAT.EXE (or the 16-bit version, which is called DISCAT16.EXE), you will see a dialog box with an edit box where you can enter a path name. After pressing the "Scan" button, the selected directory and all of its subdirectories will be traversed. Then you are asked to choose a name for the datafile which is to be used for storing the results. Apart from seeing confirmation in the form of a directory count, this application is rather boring...

Known problems: None, but you need to make sure that your system meets the requirements of either DISCAT.EXE or DISCAT16.EXE.

The DISCAT.EXE application is a 32-bit application which requires Windows 95 (or Windows NT, or Win32s, but neither of those two has been tried). In addition, you need to have two of the DLLs which come with Microsoft Visual C++ version 4.1: MFC40.DLL and MSVCRT40.DLL (these contain the MFC library, and the runtime library, respectively).

Alternately, you can use DISCAT16.EXE, which runs on either Windows 95 or Windows 3.1(1). This application requires the file MFC250.DLL in the system directory (usually C:\WINDOWS\SYSTEM) to start properly.

These DLL requirements are caused by the fact that the sample programs and the MetaKit libraries are distributed in the smallest possible format, which happens to be as MFC extension DLLs. As a result, you need to supply the MFC libraries yourself. Considering the fact that MetaKit is currently only useful in combination with MFC, this should not be a problem.

Files in EXAMPLES\DISCAT:
CATALOG.H, CATALOG.CPP - Creates a disk catalog object using MetaKit
DISCAT.H, DISCAT.CPP - Main MFC application code
DISCAT.MAK, DISCAT.MDP - MSVC 4.0 project makefiles for Win32
DISCAT.DEF, DISCAT.RC - Linker definitions, Application resources
DISCAT16.MAK - MSVC 1.52 project makefile for Win16
RESOURCE.H - Resource symbol definitions
STDAFX.H, STDAFX.CPP - Used for precompiled headers RES*.* - Application resources

DUMP.EXE

This application is a generic utility for the MetaKit library.

What it does: Dump is a utility program which displays the contents of a MetaKit datafile on standard output. The output shows all the information which is stored in the datafile, which can be quite useful while developing and debugging applications based on MetaKit.

What it illustrates: Dump is a so-called "32-bit console application", this is the modern version of the classical MS-DOS command-line executables. It shows how you can extract structural information from a MetaKit datafile without knowing anything about it. The basic trick is to simply open the file, and then to step through each of the properties (using the members NumProperties and NthProperty). The executable file is tiny, because both MetaKit and MFC have been linked as DLLs.

How to use it: To dump a datafile, you can start DUMP.EXE from the MS-DOS prompt with the filename as argument (or drop a file on it in the Explorer). Since datafiles can store arbitrarily nested data structures, you may need to examine the output to understand how the information is presented on standard output. Dump adds quite a bit of detail to guide you (including all the property names). Basically, the data structure is traversed recursively, with indentation added to reflect the current nesting level. There are a number of command-line options to modify the default output style:

-s List all tables and subtable, but omit the actual data values.
-d List all tables and subtables, including a low-level data dump.
-f Only list all the fields (properties) and their structure.

-w Wait for the RETURN key just before exiting.

You can redirect the output to file, just like any other console application.

Known problems: None. Note that this is a 32-bit application which requires Windows 95 (or perhaps Windows NT or Win32s), and that the MetaKit DLLs are used.

Files in EXAMPLES\DUMP:
DUMP.CPP - Dump main program
DUMP.MAK, DUMP.MDP - MSVC 4.0 project makefiles for Win32

STRUCT.EXE

This application highlights some low-level aspects of the MetaKit library.

What it does: Struct is a little utility which displays the data structure of any MetaKit datafile on standard output. The output uses a text-mode graph to display the data structure, which can be useful to determine what information is stored in a specific MetaKit datafile.

What it illustrates: For demonstration purposes, Struct has been compiled as a real-mode MS-DOS utility program. It shows how you can extract structural information from a MetaKit datafile without knowing anything about it. The sample code includes a general "StructureMap" class to generate text-based graphs. The size of the STRUCT.EXE program demonstrates how small a fully self-contained application using MetaKit (and a bit of MFC) can be.

Note that although Struct must respect the 640 K memory limit imposed by MS-DOS, you can still use this program to examine the structure of files of any size. The reason for this is that MetaKit implements on-demand loading, and that Struct never accesses the actual data itself.

How to use it: To examine the structure of a datafile, simply run STRUCT.EXE from the MS-DOS prompt with the filename as argument. There are a few command-line options to control the output format, these are listed when you start Struct without a filename. They are:

-d Show a linear description instead of the default tree.
-t Show a linear description of the structure, omitting the property names.
-c Show column structure instead of the deafult tree structure (advanced).

You can redirect the output to file, just as with any other MS-DOS program.
Known problems: None.

Files in EXAMPLES\STRUCT:
STRUCT.CPP - Struct main program
STRUCT.MAK - MSVC 1.52 project makefile

CATSEND.EXE

This application is a simple TCP/IP client based on Winsock.

What it does: CatSend allows you to select a datafile and to transmit its contents over a TCP/IP connection. Although CatSend can handle any MetaKit datafile, only the catalog files created by DisCat will make sense to the receiving program called "CatRecv" (see below). By altering the address shown on the screen, you can transfer the data across the network to any other machine on Internet running CatRecv... this is only a demonstration of what can be done, this serves no useful purpose whatsoever.

What it illustrates: CatSend illustrates how to work with MetaKit datafiles of which the structure is unknown or only partially known. Since CatSend doesn't care, this program will continue to work even if

DisCat and CatRecv are modified to use a different data structure on file.

How to use it: CatSend only makes sense in combination with Catrecv and DisCat. To try it out, you should first create a catalog file with DisCat which you can then pick up with CatSend. Note that like any typical client-server application, the CatRecv server application must be running before using CatSend to open a connection. Apart from selecting a datafile and transmitting it to CatRecv, the CatSend application is pretty boring. The "interesting" source code is less than 10 lines long...

Known problems: None, but since CatSend uses Winsock, you need to have a correctly installed version of the TCP/IP protocol stack (such as the one that comes with Windows 95).

You do not need an active network connection or any network hardware, since every TCP/IP installation can select local communication by using "localhost" as a host name (i.e. IP address 127.0.0.1). If you obtained this package from Internet via FTP, then you should be all set, since FTP access implies that you have a properly working TCP/IP configuration.

Files in EXAMPLES\CATSEND:
CATSEND.H, CATSEND.CPP - Main MFC application code
CATSEND.MAK, CATSEND.MDP - MSVC 4.0 project makefiles for Win32
CATSEND.RC - Application resources
MYDLG.H, MYDLG.CPP - Main dialog
RESOURCE.H - Resource symbol definitions
STDAFX.H, STDAFX.CPP - Used for precompiled headers
RES*.* - Application resources

CATRECV.EXE

This application is a simple TCP/IP server based on Winsock.

What it does: When CatRecv is started, is enters a loop waiting for the CatSend client application to connect. Each new connection sets up its own window to displays the contents of a catalog tree sent by CatSend. You can examine the catalog tree and verify that it actually contains what the DisCat sample program previously saved on file.

What it illustrates: CatRecv is pretty neat. Apart from illustrating how to transfer a complex MetaKit data structure over a network connection, it shows off the new tree and list controls now available in Windows 95 with MFC 4.0, and it demonstrates how to design a little TCP/IP network server. The catalog window uses an Explorer-like tree interface to allow you to easily examine a catalog tree structure, and it didn't even take a lot of work to create it in C++. Since this is mainly intended as a demonstration of the use of MetaKit, some more fancy user-interface features, such as sortable file lists and splitter views were not included, although this is probably quite easy to add.

How to use it: As it is, CatRecv only makes sense in combination with CatSend and DisCat. To try it out, you should first create a catalog file with DisCat which you can then pick up with CatSend. Note that like any typical client-server application, the CatRecv server application must be running before using CatSend to open a connection. Whenever a new connection is opened and a catalog object is received, CatRecv will open a new window with the contents of that catalog.

This sample application uses a fixed TCP/IP port for its communication, but you can change the default port number to any value you like in the opening dialog. Just make sure to use the same port number in CatRecv, or you won't be able to connect.

Known problems: None, but since CatRecv uses both the new Windows common controls and Winsock, you can only run this application on Windows 95, and you need to have a correctly installed version of the TCP/IP protocol stack (such as the one that comes with Windows 95 itself).

You do not need an active network connection or any network hardware, since every TCP/IP installation can select local communication by using "localhost" as a host name (i.e. IP address 127.0.0.1). If you obtained this package from Internet via FTP, then you should be all set, since FTP access implies that you have a properly working TCP/IP configuration.

Files in EXAMPLES\CATRECV:
CATRECV.H, CATRECV.CPP - Main MFC application code
CATRECV.MAK, CATRECV.MDP - MSVC 4.0 project makefiles for Win32
CATRECV.RC - Application resources
MYDLG.H, MYDLG.CPP - Opening dialog
RESOURCE.H - Resource symbol definitions
STDAFX.H, STDAFX.CPP - Used for precompiled headers
TREEDLG.H, TREEDLG.CPP - Contains the interesting pieces of this application
RES*.* - Application resources

FTPCAT.EXE

This tiny application does not illustrate any new aspects fo MetaKit, but it may be useful in its own right, especially with the CatFish application described hereafter.

What it does: FtpCat creates a catalog file of a directory tree, just like DisCat. In this case, however the directory listings are obtained from an FTP server anywhere on the network. Since FtpCat has no browse facility, you will have to use CatSend & CatrRecv (or Dump) to examine the contents of the resulting catalog.

What it illustrates: FtpCat is a 32-bit console application, which is dynamically linked to both MetaKit and MFC. FtpCat is a bit peculiar, in that it uses the FTP.EXE application which is part of Windows 95 to take care of all networking (after all, MetaKit is not an introduction on how to build ftp clients). There are no ftp-related routines, nor even any Winsock calls, in FtpCat. There is a fair amount of tricky code in here to juggle two pipes which are used to control the child process (ftp.exe).

How to use it: FtpCat is very simple to use: simply call it with an URL as argument, and it will open an ftp connection and retrieve the listings of the specified directory and all its subdirectories. If you do not supply a second parameter, the catalog will be stored in a file called "ftpcat.dat". As with DisCat, you can then use CatSend & CatRecv to examine the catalog contents.
If you're connected to Internet, you could try "ftpcat ftp://ftp.winsite.com/pub/pc/win95/programr" (this is one the areas where new distributions of the MetaKit library are posted). The "ftp://" prefix is optional, and you may add a "user:password@" prefix to the site to login under a specific name. This allows you to produce a catalog of protected areas, for example with "ftpcat user:passwd@mysite.org/mydir". Please be aware of the fact that some sites on internet are huge, and that a command such as "ftpcat ftp.winsite.com" which uses "/" as default path may take some time to complete, depending on the speed of your connection.

Known problems: None, but there are some limitations in this sample application:

1. The "site:port" notation is not implemented, you can only connect to standard ftp servers.
2. This version does not commit intermediate results to file. If you cancel it, nothing is saved.
3.

Files in EXAMPLES\FTPCAT:
FTPCAT.CPP - FtpCat main program
FTPCAT.MAK, FTPCAT.MDP - MSVC 4.0 project makefiles for Win32

CATFISH.EXE

CatFish is the flagship of these sample applications. It is a very fast disk catalog browser which can easily deal with the huge file collections often found on current hard disks and CD-ROM software collections.

What it does: CatFish allows you to create disk catalogs by scanning disks, either entirely or from a

specified directory, and to browse through the directories in each catalog to list the files, their sizes, and their modification dates. In addition, CatFish displays statistics about each directory, such as the total number of sub-directories, files, and kilobytes in each (including all nested entries). It also tells you the date of the most recent file in each directory (even if that file is inside a sub-directory). There is a flexible search facility to find files by (partial) name, size, and/or date. Finally, CatFish can be used as an application/document launcher - a bit like the File Manager (or the Explorer in Windows 95).

The data files used by CatFish use the same format as those created by DisCat and FtpCat. You can use FtpCat to create FTP server catalogs - and then use CatFish to browse through them.

What it illustrates: CatFish is a 16-bit Windows application (it runs on Windows 95, and probably Windows NT, as well). The freeware version has been built with the static library version of MetaKit and is fully self-contained. The size of the executable file illustrates how small complete applications can be when using MetaKit for persistent storage. The makefile included with the sources are configured for dynamic linkage, which will produce an even smaller executable.

CatFish uses a carefully chosen data structure with a high "locality of reference", which generates very compact catalog files and which makes browsing extremely fast - even for full CD-ROM catalogs and large FTP archive server catalogs (created with FtpCat).

There are several goodies in this sample application. Among other things, CatFish demonstrates:

- How to use a sorted view subset (selection) of MetaKit in a CListBox (MFC)
- How to implement customizable sort order for a listbox, with clickable headers
- How to scan directories recursively - in Win16 / Win32 / Win95 (long filenames)
- How to quickly display huge lists - using the owner-draw listbox control
- How to simplify columnar list formatting - using the MS Sans Serif font
- How to implement the ellipsis (...) for text strings which are too long to show
- How to deal with international dates, times, and numbers - but only a little bit
- How to launch applications and/or documents - using ShellExecute()
- How to make a modal dialog main window deal with menus and accelerators

How to use it: This sample application is also separately distributed as freeware, and includes more of the standard features you would expect than the other sample applications in this library. Using CatFish is simple: set up one or more catalog files (which will be stored in the same directory as the executable) and start browsing by clicking on the various lists in the main window. To descend into a subdirectory, double-click on its name. To go back, double-click on one of the parent directory names.

To search for a file, you can use the Edit / Find menu entry or use keyboard shortcuts, such as CTRL+F (find), F3 (find next), SHIFT+F3 (find previous).

If you want to use FtpCat in combination with CatFish, make sure that you give all catalog files the extension ".cf4" and that you place them in the same directory as the CatFish utility itself. That's all.

Known problems: None. As a 16-bit program, CatFish can handle up to some 3000 files per directory.

Files in EXAMPLES\CATFISH:
CATFISH.H, CATFISH.CPP - Main MFC application code
CATFISH.MAK - MSVC 1.52 project makefile for Win16
CATFISH.DEF, CATFISH.RC - Linker definitions, Application resources
PICKDIR.H, PICKDIR.CPP - Directory picker
RESOURCE.H - Resource symbol definitions
SCANDISK.H, SCANDISK.CPP - Creates a disk catalog object using MetaKit
SETUPDLG.H, SETUPDLG.CPP - Setup dialog
STDAFX.H, STDAFX.CPP - Used for precompiled headers RES*.* - Application resources