CHAPTER 22

Tclkit and Starkits

Tclkit is a version of the Tcl/Tk interpreter that is designed to make packaging and deployment of Tcl applications easy. Tclkit includes Tcl/Tk, [incr Tcl], the Metakit database, and TclVFS. A Starkit is a special file that contains all the scripts and supporting files you need for your Tcl application. This chapter describes how to package and deploy your application as a Starkit.

Tclkit was created by Jean-Claude Wippler as a way to make deploying Tcl applications easier. Tclkit is an extended Tcl interpreter that includes the Metakit database, the [incr Tcl] object-oriented system, and a Virtual File System (VFS). The database is cleverly stored as part of the Tclkit application itself, and the VFS interface is used to make the database look like a private filesystem. Tclkit puts all the scripts normally associated with Tcl and its extensions into this database. The result is a self-contained, single file distribution of Tcl that includes extensions for your GUI, object-oriented programming, a database, and a few other goodies.

Metakit is a fast, transactional database with a simple programming API. Like Tcl, Metakit is a compact, efficient library designed to be embedded into applications. The Tcl interface to Metakit gives you a simple, easy way to manipulate persistent data. Although you do not have to program Metakit directly when using Starkits, this Chapter does provide a short introduction to using Metakit to store data for your application.

A Starkit is a Metakit database file that stores your application. The VFS interface makes this transparent. Tclkit processes the Starkit just like *tclsh* or *wish*, and your application doesn't even have to know it is packaged inside a Starkit.

The original Tclkit used an early version of VFS created by Matt Newman. TclVFS was ported to the Tcl core in version 8.4.1 by Vince Darley. Today you can build Tclkit using unmodified Tcl sources. The ActiveTcl distribution includes Metakit, TclVFS and tools to create Starkits, too.

Getting Started with Tclkit

Using Tclkit is easy. Just copy the version for your platform (e.g., Linux, Windows or Solaris) into a convenient location under the name *tclkit* (or *tclkit.exe* on Windows.) The CD-ROM has builds for lots of platforms, and you can find more at the Tclkit home page:

```
http://www.equi4.com/tclkit
```

You can use the *tclkit* application just like *tclsh*. Run with no arguments, it prints a prompt and you can type Tcl commands interactively. If you pass a file argument, then it sources that file just as *tclsh* would. To use *tclkit* like *wish*, you must add this to your scripts:

```
package require Tk
```

Although you can use *tclkit* to source `.tcl` files, *tclkit* is normally used to interpret Starkits, which have a `.kit` suffix. On UNIX, Starkits use the `#!` header to associate themselves with *tclkit*. Make sure that *tclkit* is in a directory named in your `PATH` environment variable. On Windows, you can associate *tclkit.exe* with the `.kit` extension. Mac OS X behaves like UNIX (yay!). On Mac Classic systems you can use the `File Source` menu to source `.kit` files. Creating Starkits is described on page 352.

Inside a Starkit

Tclkit uses the Virtual Filesystem extension to make records in a Metakit database look like files and directories to your application. Through a simple packaging step described shortly, you can easily put all of the Tcl scripts and other supporting files that make up your application into a single database file. The Virtual Filesystem (VFS) extension lets you transparently access these files through the regular file system interface (e.g., `open`, `gets`, `source`, even `cd`.)

A *Starkit* is a Metakit database that stores an application. The great thing about a Starkit is that it is a single file so it is easy to manage. There is no need to unpack files or run an installer to set things up. Instead, you can distribute your application as two files: the Tclkit interpreter and the Starkit file. Both of these embed a virtual file system that include all the bits and pieces needed for Tcl/Tk and your application. The Tclkit file is platform-specific because it contains Tcl and all the other extensions in a compiled form. There are pre-compiled Tclkits for Windows, Macintosh, and many flavors of Unix. The Starkit file is platform-independent. You can use it with the appropriate Tclkit interpreter on different platforms.

Deploying Applications as Starkits

The key benefit of Tclkit and Starkits is easy deployment. Users just copy *tclkit* and your Starkits onto their system; there is no special installation step. You can even have different versions of *tclkit* and they don't interfere with each other. If users get tired of your application, they just remove the files.

Creating Starkits is made easy with the *sdx* application, which was created by Steve Landers and Jean-Claude Wippler. You organize your collection of application scripts, data files, binary graphics, and online documentation into a file system directory structure. Then you use *sdx* to wrap that into a Starkit. Creating your own Starkits is described on page 352.

You can include binary extensions in a Starkit and dynamically load them. The `load` command automatically copies the shared library out of the VFS to a temporary location, and loads the library from that location. The temporary file is necessary because the host OS cannot find the library inside the Starkit. Binary extensions make the Starkit platform-specific, but it is possible to put libraries for different platforms into the Starkit. For example, the `kitten.kit` Starkit includes extensions for Windows, Linux, and Solaris.

You can combine Tclkit and a Starkit into a *Starpack*. The advantage of this is that it reduces deployment to a single file. The main drawback is that the Starpack file is relatively large, and it is platform-specific. Use *sdx* to create Starpacks as described later.

The Starkit archive contains a growing collection of Starkits that include applications, games, development tools, a Wiki, tutorials and documentation bundles. There is a copy of the archive on the CD-ROM, and its home page is:

```
http://mini.net/sdarchive/
```


Virtual File Systems

The key concept in Tclkit and Starkits is the virtual file system (VFS). You may be familiar with the file system interface inside a Unix operating system that makes everything look the same (files, tape drives, network sockets, pipes). The nice thing about Unix is that a system programmer can use the same APIs to access all of these things. The goal of the Tcl VFS interface is similar in spirit: use the regular Tcl file system interface to make things like embedded databases, FTP servers, and zip files available to the Tcl programmer. The VFS layer in Tcl 8.4 is implemented below the Tcl C APIs for file system access (e.g., `Tcl_CreateChannel`, `Tcl_FSDeleteFile`). The result is that scripting commands (e.g., `open`, `file`, `glob`) and any C extensions that use these APIs automatically access any Virtual File Systems that are part of the Starkit.

The virtual file system is *mounted* on a regular file; by default it is mounted on the Starkit. For example, if the Starkit is named `foo.kit`, and its virtual file system contains a file named `main.tcl`, then it is visible to the Tcl application as `foo.kit/main.tcl`. The VFS can contain a whole directory structure (e.g., `foo.kit/lib/httpd.tcl` or `foo.kit/htdocs/help/index.html`.)

The next section explores some simple Starkits and their file system structure. The main idea is that the Starkit file itself is the root of the virtual file system hierarchy, and everything in the virtual file system is visible to Tcl via the regular scripting commands. If the VFS supports it, you can create and write files as well as read them.

Tclkit includes the TclVFS extension that exposes the ability to implement new file systems in Tcl. Ordinarily you do not need to use the `vfs` API directly when using a Starkit. However, the TclVFS project has created a number of VFS implementations that let you access web sites, FTP sites, zip files, tar files, and more through the filesystem interface. Tclkit does not include all of these, but you can get them as part of the TclVFS extension. Its home page is

```
http://sourceforge.net/projects/tclvfs
```

Accessing a Zip File Through a VFS

Tclkit includes a *zipvfs* package that lets you mount a compressed ZIP file archive and read its contents. This is currently limited to read-only access. Example 22–1 uses the `vfs::zip::Mount` command to set up the VFS access. If you use other VFS types supplied by the TclVFS extension, you will find that each supplies its own `vfs::`*`vfs_type`*`::Mount` API:

Example 22–1 Accessing a Zip file through a VFS

```
package require vfs::zip
=> 1.0
# Mount the zip file on "xyz"
vfs::zip::Mount c:/downloads/tclhttpd343.zip xyz
=> filecb15a8
# Examine the contents
glob xyz/*
=> xyz/tclhttpd3.4.3
# Open and read file inside the zip archive
set in [open xyz/tclhttpd3.4.3/README]
=> rechan16
gets $in
This HTTPD is written in Tcl and Tk.
```

Using *sdx* to Bundle Applications

Sdx, which stands for Starkit Developer eXtension, is an application that you run from the Unix, Windows, or MacOS command line to create and manipulate Starkits. It is itself a Starkit, of course. The *sdx* application is on the CD-ROM, and you can find a link to it from the Starkit home page:

```
http://www.equi4.com/starkit/
```

Creating a Simple Starkit

Creating a Starkit amounts to creating a directory structure that contains the files you need, and then wrapping them up with *sdx*. Create files under *`kitname`*`.vfs`, and wrap them into the *`kitname`*`.kit` Starkit with:

```
sdx wrap kitname.kit
```

In simple cases, *sdx* will create the directory structure for you. For example, if you have a self-contained Tcl script called `hello.tcl`, then you can turn it into a Starkit like this:

```
sdx qwrap hello.tcl
```

The `qwrap` operation (i.e., "quick wrap") creates a new Starkit, `hello.kit`, that includes the original `hello.tcl` script organized into a virtual file system hierarchy with some additional support files. You run the Starkit like this:

```
tclkit hello.kit
```

On Unix systems you can also execute the Starkit directly. The file uses the `#!` syntax to specify that *tclkit* should run the file. On Windows, you can achieve the same effect by associating *tclkit.exe* with files that end in `.kit`.

Examining a Starkit

There are two ways to look at a Starkit. You can get a listing of the files with the `sdx lsk` operation, or you can use `sdx unwrap` to extract the files from the Starkit into a *`kitname`*`.vfs` directory. Example 22–2 shows the `lsk` output for `hello.kit`. The dates are in `YY/MM/DD` format:

Example 22–2 The output of `sdx lsk hello.kit`.

```
hello.kit:
                       dir  lib/
        67  02/11/08 12:07  main.tcl
hello.kit/lib:
                       dir  app-hello/
hello.kit/lib/app-hello:
        43  02/11/08 12:10  hello.tcl
        72  02/11/08 12:07  pkgIndex.tcl
```

Standard Package Organization

The `qwrap` operation turns the `hello.tcl` script into the `app-hello` package. If necessary, *sdx* adds a `package provide app-hello 1.0` command to the `hello.tcl` script. It also creates a short `main.tcl` script that initializes the Starkit system and invokes `hello.tcl` by doing a `package require`. Example 22–3 shows `main.tcl`:

Example 22–3 The main program of a Starkit.

```
package require starkit
starkit::startup
package require app-hello
```

When you run the Starkit, its Metakit database is mounted into a Virtual File System that is visible to the Tcl application. Tclkit sources the `main.tcl` script it finds in the VFS. The `starkit::startup` procedure updates the

auto_path to contain the Starkit's lib directory, so any packages stored there are available to the package mechanism. By convention, the application is put into a package with the name app-*kitname*. Example 22–4 shows the pkgIndex.tcl, which causes the package require app-hello command to source hello.tcl.

Example 22–4 The pkgIndex.tcl in a Starkit.

```
package ifneeded app-hello 1.0 \
    [list source [file join $dir hello.tcl]]
```

The dir variable is set by the package mechanism to be the directory containing the pkgIndex.tcl file. That the lib directory happens to be inside the virtual file system is completely transparent to the package mechanism. The package mechanism is described in more detail in Chapter 12.

Creating a Starpack

A Starpack contains a copy of Tclkit and your Starkit. Use *sdx* to create Starpacks. The -runtime flag specifies which Tclkit application you want to merge with your Starkit. For example, to build a Windows Starpack out of our hello.tcl application:

```
sdx wrap hello.kit -runtime tclkit-win32.exe
```

To build a Starkit for Linux, use the appropriate runtime:

```
sdx wrap hello.kit -runtime tclkit-linux-x86
```

There are 4 variations of the Windows Tclkit. One option uses *zlib* to automatically compress Tclkit and the Metakit database. These have .upx in their name. The other creates a console-mode application that does not include Tk. These have -sh in their name. The smallest Tclkit, tclkit-win32-sh.upx.exe, is only 450 K. Even tclkit-win32.upx.exe is only 907 K, so you really can create complete applications that fit easily onto a floppy disk!

The auto-compress variation is also available on the Linux x86 builds as the tclkit-linux-x86.upx.bin runtime file. Check the Tclkit home page for the latest set of Tclkit builds:

```
http://www.equi4.com/tclkit
```

Exploring the Virtual File System in a Starkit

Example 22–2 introduces the standard, recommended VFS structure for a Starkit that makes everything into a package, even the main application. However, in this section we are going to show a Starkit without packages in order to get a feel for how the VFS works. For example, instead of doing the package require hello, the main.tcl script of Example 22–3 could source the hello.tcl file directly:

```
source hello.kit/lib/app-hello/hello.tcl
```

However, this only works if you are in the directory containing the `hello.kit` file.

Use `starkit::topdir` to find things in the Starkit Virtual File System.

The `starkit::topdir` variable is set by `starkit::startup` to be the file name of the Starkit, which is also the root of the Virtual File System inside the Starkit. The value of `starkit::topdir` is an absolute pathname, so it is always valid. Example 22–5 shows a Starkit that manipulates its virtual file system.

Example 22–5 A Starkit that examines its Virtual File System.

```
package require starkit
starkit::startup

puts "Contents of VFS before"
foreach f [glob [file join $starkit::topdir *]] {
   puts "[file size $f] $f"
}
puts "Reading data file"
set in [open [file join starkit::topdir data]]
set X [read $in]
puts $X
close $in
set out [open [file join $starkit::topdir data.new w]]
puts $out $X
close $out
puts "Contents of VFS after"
foreach f [glob [file join $starkit::topdir *]] {
   puts "[file size $f] $f"
}
```

Create the Starkit by putting the code in Example 22–5 into a file named `main.tcl` in the `write.vfs` directory. Then use *sdx* as shown in Example 22–6:

Example 22–6 Creating a simple Starkit.

```
# These are UNIX shell commands
mkdir write.vfs
cp 22_5.tcl write.vfs/main.tcl
sdx wrap write.kit
tclkit write.kit
```

If you run the `write.kit` file more than once you will notice that the `write.kit/data.new` file does not persist between runs. This is because, by default, the Metakit database is modified in main memory and it is not written out to the Starkit file. If you want to store files long term, use the `-writable` flag to *sdx*:

```
sdx wrap write.kit -writable
```

Creating `tclhttpd.kit`

The Tcl Web Server, TclHttpd, has its source tree organized so you can run the server without any installation steps. This makes it very easy to put into a Starkit. For our first version, which we will refine later, all we need is a copy of the TclHttpd source code and a copy of the Standard Tcl Library, *tcllib*. I used the `tcllib1.3` directory that was installed in the main lib directory of my desktop Tcl environment, and the `tclhttpd3.4.3` source distribution. Example 22–7 shows the contents of the `tclhttpd.vfs` directory:

Example 22–7 The contents of the `tclhttpd.vfs` directory, version 1.

```
main.tcl
tclhttpd3.4.3/bin/httpd.tcl
tclhttpd3.4.3/bin/httpdthread.tcl
tclhttpd3.4.3/bin/tclhttpd.rc
tclhttpd3.4.3/lib/ (lots of files)
tclhttpd3.4.3/htdocs/ (lots of files)
tcllib1.3 (copy of /usr/local/lib/tclib1.3)
```

Example 22–8 shows the short `main.tcl` script used to start up the Starkit. The first two lines are common to all Starkits. The `starkit::autoextend` command is used to add the `tcllib1.3` directory to the `auto_path` so the Standard Tcl Library packages are available. The last line uses `starkit::topdir` to find the TclHttpd startup script, `bin/httpd.tcl`.

Example 22–8 The main program for the TclHttpd Starkit, version 1.

```
package require starkit
starkit::startup
starkit::autoextend [file join $starkit::topdir tcllib1.3]
source [file join $starkit::topdir tclhttpd3.4.3/bin/httpd.tcl]
```

The Starkit is created and used as shown below, assuming `tclhttpd.vfs` is in the current directory. Note that command line options are passed through, so you can also use this Starkit to host an `htdocs` directory outside the Starkit. If you don't specify one, the `htdocs` tree inside the Starkit is used:

```
sdx wrap tclhttpd.kit
tclkit tclhttpd.kit -port 8080 -docRoot /my/htdocs
```

The standard structure introduced in Example 22–2 organizes packages under a `lib` directory. By convention, the version numbers are dropped from the package directory names. Because everything is self contained, there really isn't any need to have explicit version numbers in the directory names. The file system for the second version of `tclhttpd.kit` is shown in Example 22–9.

Example 22–9 Contents of the `tclhttpd.vfs` directory, version 2.

```
main.tcl
bin/httpd.tcl
bin/httpdthread.tcl
bin/tclhttpd.rc
lib/tclhttpd/pkgIndex.tcl
lib/tclhttpd/*.tcl (lots of files)
lib/tcllib/pkgIndex.tcl
lib/tcllib/* (lots of subdirectories)
```

The `main.tcl` file is shown in Example 22–10. There is no need to adjust the `auto_path` because `starkit::startup` ensures that the `lib` directory is on it.

Example 22–10 The main program for the TclHttpd Starkit, version 2.

```
package require starkit
starkit::startup
source [file join $starkit::topdir bin/httpd.tcl]
```

One of the first things I noticed about the `tclhttpd.vfs` was that *tcllib* took up far more space than the rest of TclHttpd. TclHttpd only uses a few of the many modules in *tcllib*. I ended up only adding the modules I needed in order to keep the Starkit smaller. Another way to solve this problem is to use the `tcllib.kit` Starkit that can be shared among applications. Creating shared Starkits is the topic of the next section.

Creating a Shared Starkit

Starkits can be used to create modules that are shared by other applications. For example, the `kitten.kit` Starkit contains about 50 popular extensions, and several of them are binary extensions. It is over 4 MB in size, and so it is a great candidate for sharing. You can find `kitten.kit` on the CD-ROM or in the Starkit archive. By organizing each shared module into a Starkit with the appropriate structure, it is a simple matter to share them.

Whenever a Starkit is sourced, *Tclkit* mounts its VFS and looks for its `main.tcl` file. This is true for shared Starkits as well as the main Starkit of an application. If `main.tcl` calls `starkit::startup`, then the `lib` directory in the VFS is automatically added to the `auto_path`. Any libraries organized under `lib` will be automatically accessible to the application that sourced the Starkit.

You can add a little logic to make your package behave differently if it is run as the main Starkit or sourced into another application. For example, this is done in the *tcllib* Starkit, which starts a stand-alone Wiki that describes the Standard Tcl Library APIs if run as its own Starkit. Otherwise it just sets up *tcllib* to be shared by the main application. Example 22–11 shows the `main.tcl` of `tcllib.kit`. It has to explicitly add the `tcllib` directory to the `auto_path` because it has both a `lib` and `tcllib` directory in its VFS:

Example 22–11 The Standard Tcl Library Starkit `main.tcl` file.

```
package require starkit
if {[starkit::startup] eq "starkit"} {
    # Do application startup
    package require app-tcllib
} else {
    # Set up to be used as a library
    set vfsroot [file dirname [file normalize [info script]]]
    lappend auto_path [file join $vfsroot tcllib]
}
```

Another side effect of `starkit::startup` is to set `starkit::topdir`. However, this variable is only set once. If you source other Starkits that call `starkit::startup`, then the `starkit::topdir` value is not disturbed.

This behavior changed in Tclkit 8.4.2. In earlier versions, `starkit::topdir` was set by each Starkit, so you had to worry about saving its value if you loaded other Starkits. If you source `tcllib.kit` and cannot `package require` its packages, check its `main.tcl`. If it uses `starkit::topdir` in the non-Starkit case, then it is an older version. Simply unwrap it, make its `main.tcl` look like Example 22–11, and wrap it back up to fix the problem.

The `starkit::startup` procedure determines the environment of the application by making a series of tests against the script environment. Its return value helps your `main.tcl` script distinguish between starting out as the main Starkit, or being loaded into another Starkit as a library. Table 22–1 lists the return values of the `starkit::startup` procedure in the order they are checked:

Table 22–1 Return values of the `starkit::startup` procedure.

`starpack`	The Starkit was bundled with *tclkit* to make a Starpack.
`starkit`	The Starkit was run by itself.
`unwrapped`	The Starkit was run out of its unpacked vfs directory.
`tclhttpd`	The Starkit was sourced into TclHttpd.
`plugin`	The Starkit was sourced in the browser plugin.
`service`	The Starkit was run in an NT service.
`sourced`	The Starkit was sourced by another Starkit.

The easiest way to organize your shared Starkits is to put them into the same directory. Example 22–12 shows how the TclHttpd Starkit is modified to load the *tcllib* Starkit from the same directory.

Example 22–12 The main program for TclHttpd Starkit, version 3.

```
package require starkit
starkit::startup
set dir [file dirname $starkit::topdir]
if {![file exists [file join $dir tcllib.kit]]} {
   puts stderr "Please install tcllib.kit in $dir"
   exit 1
}
source [file join $dir tcllib.kit]
source [file join $starkit::topdir tclhttpd/bin/httpd.tcl]
```


Metakit

This section provides a short overview of the Metakit database that is used by Starkits to store their data. You do not need to program Metakit directly to use Starkits because of the transparent VFS interface. However, Metakit is an easy-to-use database that provides more power than storing data in flat files, but not as much power (or overhead) as a full SQL database engine. Metakit has a simple, flexible programming API and an efficient implementation. By storing your application data in a Metakit table, you can have persistent data that lives with your application. You can store the data in a file separate from your application, or right inside the application Starkit itself.

This Chapter gives a few introductory examples and explains some of the other features that are available. This Chapter does not provide a complete reference. The following URLs are excellent guides to the Tcl interface for Metakit. The first URL is also on the CD as `sdarchive/doc/mk4dok.kit`.

```
http://www.equi4.com/metakit/tcl.html
http://www.equi4.com/metakit/wiki.cgi/mk4tcl
http://www.markroseman.com/tcl/mktcl.html
```

Metakit Data Model

The Metakit data model is table-oriented. A *view* is like a table with rows of values. Each row in a view has an *index*, which is an integer that counts from 0. The elements (i.e., columns or fields) of a row are called *properties*. A property might itself be a view, which leads to nested views (i.e., nested tables). All the rows in a view have the same properties, and the properties of a view can be changed dynamically. You can directly relate (view, row, property) to (table, row, field) when thinking about Metakit views.

A Metakit data file has one or more views within it. When you open a Metakit file, you specify a *tag*. Views are specified as *tag.view*. Row N of a view is specified as *tag.view!N*. Such a position within a view is called a *cursor*, and there are operations to create cursor variables and move them through a view. If a property is a nested view, then you can specify a row in the nested view with *tag.view!N.subview!M*.

Examining a Metakit Database

Our first exercise is to open up a Starkit and look at the Metakit database views inside. The `mk::file` command implements several operations. The `open` operation opens a database and associates it with a tag. The `views` operation lists the views in the database identified by the tag. The `close` operation commits any outstanding modifications to the database. The other `mk::file` operations are used to control the commit behavior and to save or restore the database to an external file. Example 22–13 illustrates how to open a Metakit database and examine the views it contains:

Example 22–13 Examining the views in a Metakit database.

```
package require Mk4tcl
=> 2.4.8
mk::file open tclhttpd tclhttpd.kit
=> tclhttpd
mk::file views tclhttpd
=> dirs
```

The `mk::view` command has several operations to inspect and manipulate views. The `layout` operation queries or sets the properties of a view. Given only a view, the `layout` operation returns the properties defined for the view. Each property has a type, and nested views are represented as a nested list of the property name and its list of properties. Given a set of properties, the `layout` operation defines new properties for a view. This may involve adding or deleting properties from any existing rows in the table. Example 22–14 shows the layout of the `dirs` view in a Starkit. The `files` property is a nested view, which provides a natural way to represent a hierarchical filesystem. The example gets the `name` property of `tclhttpd.dirs!0.files!0`, which is the first file in the first directory in the view:

Example 22–14 Examining data in a Metakit view.

```
mk::view layout tclhttpd.dirs
=> name parent:I {files {name size:I date:I contents:B}}
mk::view size tclhttpd.dirs
=> 48
mk::get tclhttpd.dirs!0
=> name <root> parent -1
mk::get tclhttpd.dirs!1
=> name tcllib1.3 parent 0
mk::get tclhttpd.dirs!1 name
=> tcllib1.3
mk::get tclhttpd.dirs!0.files!0 name
=> main.tcl
```

Of course, real applications will want to query views for values that have certain properties. The `mk::select` command returns the row numbers for rows

that match given criteria, or all the row numbers if no matching criteria are given. You can match on multiple properties, and there are flags that control how the match is done. For example, you can do numeric comparisons, regular expression or glob matches, and min/max comparisons.

Example 22–15 shows two forms of `mk::select`. The `KitWalk` procedure enumerates the files in a given directory, which is the view `$tag.dirs!$dir.files`. Then it queries the row indices for the `$tag.dirs` view whose `parent` property equals `$dir`, and calls itself recursively to process the child directories. `KitWalk` provides a similar function to `sdx lsk`:

Example 22–15 Selecting data with `mk::select`.

```
proc KitWalk {tag dir {indent 0}} {
    set prefix [string repeat " " $indent]
    puts "$prefix[mk::get $tag.dirs!$dir name]/"
    incr indent 2

    # List the plain files in the directory, if any

    foreach j [mk::select $tag.dirs!$dir.files] {
        puts "$prefix  [mk::get $tag.dirs!$dir.files!$j name]"
    }

    # Recursively process directories where $dir is the parent

    foreach i [mk::select $tag.dirs parent $dir] {
        KitWalk $tag $i $indent
    }
}
proc KitInit {starkit} {
    mk::file open starkit $starkit
    if {[mk::file views starkit] != "dirs"} {
        mk::file close $starkit
        error "This database is not a starkit"
    }
    return starkit        ;# db tag
}
proc KitTest {} {
    set tag [KitInit tclhttpd.kit]
    KitWalk $tag 0
}
```

Creating a Metakit View

Creating a new view is simple. Example 22–16 opens a database file `mydb.tkd` and creates a view `test` with three properties: `name`, `blob`, and `i`. If the file does not exist, then it gets created automatically. If the `test` view doesn't exist, it gets created. If it already exists, it is reformatted to have the new properties. The `name` property has the default type, which is a null-terminated string. The `blob` property is a binary value (`B`) which can store anything, including null

characters. The `i` property is a 32-bit integer (`I`). Other types include 64-bit integer (`L`), 32-bit floating point (`F`), 64-bit double-precision floating point (`D`), and null-terminated string (`S`), which is the default and needn't be specified.

Example 22–16 Creating a new view.

```
mk::file open mydb mydb.tkd
=> mydb
mk::view layout mydb.test {name blob:B i:I}
=> mydb.test
mk::file close mydb
```

The `mk::set` command sets property values, and the `mk::row` command modifies rows. Example 22–17 adds a few values to the `test` view. Note that you can insert into rows beyond the end of the view and it is automatically extended. If you only define some properties for a row, the other properties get default values. Other `mk::row` operations include `insert`, `replace`, and `delete`.

Example 22–17 Adding data to a view.

```
mk::set mydb.test!0 name hello
=> mydb.test!0
mk::get mydb.test!0
=> hello {} 0
mk::row append mydb.test "line two" 0x0 65
=> mydb.test!1
mk::view size mydb.test
=> 2
mk::set mydb.test!100 i 1234
=> mydb.test!100
mk::view size mydb.test
=> 101
```

Storing Application Data in a Starkit

Your application can create new views in a Starkit to store persistent data. Remember to wrap your application with the `-writable` flag. You can determine the name of the Starkit from `$starkit::topdir`, and then define a new view within it. Of course, remember that Starkits use `dirs` view to store files, but you can create any number of other views within your Starkits. This is illustrated in Example 22–18, which records each time the application was run in a simple `audit` view.

Example 22–18 is careful to find the existing Metakit handle that is already opened by Tclkit. The `vfs::filesystem info` command returns an alternating list of VFS names and their Metakit database handle. The example extracts the handle and saves it in the `$db` variable. This is important because opening the same Metakit file twice (for writing) can cause corruption:

Example 22–18 Storing data in a Starkit.

```
package require starkit
starkit::startup
set db [lindex [vfs::filesystem info [$starkit::topdir]] 1]
mk::view layout $db.audit {action timestamp:I}
mk::row append $db.audit "Run as pid [pid]" [clock seconds]
puts "$argv0 has been run [mk::view size $db.audit] times"
```


To test this, put this example into the `main.tcl` of a trivial Starkit. When you create the Starkit, remember the `-writable` option with *sdx*:

```
mkdir bundle.vfs
cp 22_18.tcl bundle.vfs/main.tcl
sdx wrap bundle.kit -writable-
```

Wikit and the Tcler's Wiki

The alternative to storing data in the Starkit file is to have a separate Metakit data file. This is the approach taken by *Wikit*. The `wikit.kit` file is the Wikit application, and the `wikit.tkd` file is a Metakit database file that stores all the pages in the Wiki. (Creating a new Wiki is simple, just specify a different `.tkd` file name.) The advantage of having a separate Metakit file is that you can easily maintain your application by unwrapping and wrapping your application Starkit. Otherwise, if you put the application data directly into the Starkit you have to extract it and restore it as an additional maintenance step. In that case, you must use the `mk::file save` and `load` operations to save and restore your Metakit views to a file.

A Wiki is a web site that users can easily edit using a simplified markup syntax. Wikit is a Wiki implementation in Tcl using Metakit to store pages. It can run as a stand-alone Tk application, a GGI script, as its own little web server, or embedded into another application as a documentation bundle. There is a copy of `wikit.tkd` on the CD-ROM. For example, you run a stand alone copy of the Tcler's Wiki as:

```
tclkit wikit.kit wikit.tkd
```

The live Wiki is at `wiki.tcl.tk`*, and you can find out more about Wikit at:

```
http://wiki.tcl.tk/wikit
```

More Ideas

This Chapter has provided a brief introduction to Tclkit, Starkits, and Metakit. This should be enough to help you get started creating your own Starkits and using Metakit for persistent storage. You should consult the documentation on the Web for more detailed reference material.

* http://wiki.tcl.tk is an alias for http://mini.net/tcl.

Document Bundles

The Starkit archive includes a number of documentation bundles. For example, `mk4dok.kit` is a Starkit that contains all the MetaKit documentation. These document bundles are all based on Wikit. It is very easy to create Wiki-style documentation for your application and then bundle it up as a Metakit file. You can load `wikit.kit` and your `.tkd` document bundle into your application and use the "local" Wikit interface to display your documentation. For example, the *critcl* Starkit displays its help with this simple command:

```
Wikit::init [file join $::starkit::topdir doc critcl.tkd]
```

Self-Updating Applications

The client in a client-server application is an ideal candidate for a self-updating application. The front-end client is a Starkit with some simple startup logic that connects to a server via HTTP and displays a pretty splash screen. The server, which is often based on TclHttpd, delivers code updates to the client. The client caches the code in the VFS inside the Starkit. The application is maintained on the server, and clients automatically get updated as they are used.

This scenario has the same deployment advantage as browser-based applications: you deploy a "thin-client" to desktops that rarely, if ever, changes and you update the application code on the server. In addition, this application structure lets you create a nice client front-end that uses Tcl/Tk instead of HTML, yet still have the benefit of an easy to manage server-side installation of the application code. This design pattern is being used for a number of large-scale commercial application deployments with considerable success.

A similar system is used with the Starkit archive. If you do:

```
sdx update tclhttpd.kit
```

The *sdx* application contacts the web server running the archive and checks for any updates available for the Starkit. Only the differences are transmitted, so updates are quick, and they are automatically applied to your copy of the Starkit. This should work for all the Starkits in the snapshot of the archive on the CD-ROM.

Simple Installers

In some cases you simply must install a collection of files as part of your application. It is very easy to include those files in the VFS, and then extract them into the local file system the first time your application runs. Or, you can create a traditional "installer" that unpacks the entire application from the Starkit (or Starpack).