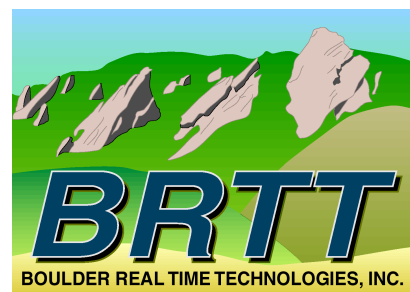


Software Development with Antelope



Revision 1.0 June, 2002



The information in this document has been reviewed and is believed to be reliable. Boulder Real Time Technologies, Inc. reserves the right to make changes at any time and without notice to improve the reliability and function of the software product described herein.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of Boulder Real Time Technologies, Inc.

Copyright © 2002 Boulder Real Time Technologies, Inc. All rights reserved.

Printed in the United States of America.

Boulder Real Time Technologies, Inc.
2045 Broadway, Suite 400
Boulder, CO 80302

Contents

CHAPTER 1	<i>Introduction.....</i>	<i>1</i>
CHAPTER 2	<i>Man pages.....</i>	<i>5</i>
CHAPTER 3	<i>Time Conversion.....</i>	<i>9</i>
CHAPTER 4	<i>Parameter Files.....</i>	<i>13</i>
CHAPTER 5	<i>Datascope.....</i>	<i>19</i>
CHAPTER 6	<i>Error Handling.....</i>	<i>27</i>
CHAPTER 7	<i>Shell Scripts.....</i>	<i>31</i>

CHAPTER 8	<i>Tcl/Tk Scripts</i>	41
CHAPTER 9	<i>Perl Scripts</i>	63
CHAPTER 10	<i>C programs</i>	75
CHAPTER 11	<i>Makefiles</i>	93
CHAPTER 12	<i>C Library Tour</i>	99
CHAPTER 13	<i>Extending Antelope</i>	117
CHAPTER 14	<i>Larger Development Efforts</i>	127
CHAPTER 15	<i>Summary</i>	133
CHAPTER 16	<i>References</i>	135

The Antelope Real Time System includes a wide variety of standard tools for real time data acquisition, system control and data archiving, along with basic data analysis tools such as event location and review. These standard tools provide the fundamental capabilities which every network operator requires. In addition, however, most operators have special requirements which are not satisfied. These requirements are often unique to the particular network and organization. BRTT generally cannot anticipate nor respond to these special needs. However, the Antelope development environment helps operators develop their own new tools which are well integrated with the standard collection.

Learning this development environment may be a bit daunting, particularly for the inexperienced programmer. However, the time spent should pay off both in the development of programs for your special needs and in the understanding of any problems which may arise in normal operations.

This guide attempts to provide a simple and gradual introduction to the many tools available. It is *not* an introduction to UNIX, to program development under UNIX, nor to programming. You should be able to follow along without much background in these areas, but you must look elsewhere for more comprehensive information. Many books provide excellent introductions to these subjects; please see the reference list for some of our favorites.

In addition to learning about UNIX and programming in general, you should be conversant with the *Antelope Datascope tutorial*, and the *Antelope Real Time System Configuration* manual. Both are distributed with Antelope, and should be found in the directory `$(ANTELOPE)/docs`.

Much of the Antelope documentation is provided in a traditional UNIX man page format. Initially, many people find man pages unhelpful, but experienced users find them indispensable. This document should provide some of the background you need to use the man pages effectively.

Antelope provides considerable documentation, including the man pages, in html format. You can look at the html version of the man pages and other documentation with your favorite browser.

```
% netscape $(ANTELOPE)/antelope.html &
```

The html format has links which allow you to move swiftly from one man page to another.

To read more about the traditional command line interface to man pages, type

```
% man man
```

It can be very productive to use the `-k` option to man to find man pages of interest.

Another approach to gaining the overview you may need to use the man pages is to skim through the reference guides. These should give you a sense of the breadth of the tools and programmatic interfaces.

The first chapters address some of the common concepts and data formats used by Antelope programs: error handling, epoch times, parameter files, and man pages. Later chapters introduce programming with scripts: plain shell scripts, and Tcl/Tk and Perl scripts.

Next, there is an introduction to using writing programs in C, and using Makefiles. This is followed with a brief tour of the important interfaces available from C programs. Finally, there is a brief discussion of using cvs to organize a source archive, and some recommendations for further reading.

There is no chapter on the contributed Matlab interface, but the document *Antelope Toolbox for Matlab* by the interface author, Kent Lindquist, is included in the docs directory of the Antelope distribution.

There are numerous example programs throughout; the source code for these examples is distributed with Antelope, under the directory `$ANTELOPE/examples`. It must be emphasized, however, that these example programs are primarily for illustration. They may be lacking essential features required in an operational program.

Some examples use a small database `demo` which is distributed with Antelope. You can find this database in the directory `$ANTELOPE/data/db/demo` in the 4.4 release.

For the user, man pages provide guidance into how to use a program. They also give some insight into what the developer was attempting to accomplish. Man pages are simple, focused, and easily accessible, with a well understood indexing system. They have various problems, like an inability to contain graphics, but they remain the primary form of documentation for Antelope and Unix.

Man pages are important to the developer not just as a means of communicating to the user, but in their own right. They are quite different from comments within the code. They provide an opportunity for the developer to step back and view the development from a different viewpoint: how to use the program (or library interface), rather than how to implement it. This review often suggests improvements and enhancements. A program is never complete until a man page has been written.

Antelope has a standard template for man pages, which closely follows the original Unix standard. The template itself is found in

```
$ANTELOPE/data/templates/manpage
```

This template has lots of comments, which should be deleted as the man page is filled out. It's often convenient to use the script `manpage` to make a local copy of the template:

```
% manpage - > example.1
```

Here, the single argument `-` causes `manpage` to remove the comments from the template. After writing one or two man pages, you probably won't need them anymore.

Fill in the synopsis section and the other sections using other man pages as examples. Though the markup is `troff/nroff` style, it's very straightforward. Here's a quick synopsis:

- `.SH "section header"`

Separates major sections. Usually, you should stick with the specified sections, in the order in which they occur in the template. Some of these sections may be empty, though.

- `.SS "sub-section header"`

Sometimes it's useful to break up the DESCRIPTION section a bit with sub-sections.

- `.LP`

left justified paragraph, the standard in man pages.

- `.nf`

- `.fi`

If you have a section which you've carefully formatted with spaces, you can surround it with `.nf` and `.fi` to preserve perhaps an indent. But the man pages fonts are not monospaced -- not all characters have the same width -- so attempts to format using spaces are generally doomed.

When you must, as in the EXAMPLE section, switch to a monospaced font like this:

```
.ft CW
.in 2c
.nf
```

Then switch back with:

```
.fi
.in
.ft R
```

The `.in 2c` indents a little, and the later `.in` reverts to the previous margin.

You may find it useful to learn the `tbl` preprocessor for making tables.

Another commonly used markup, which may occur anywhere in the text, is font changes:

- `\fI`
- `\fB`

- `\fP`

The first changes to italics, the second to bold, and the third reverts to the previous setting. Try to match up every font change with a `\fP` to revert, and use italics for emphasis, not bold.

Try to stick closely to the standard format. People are already familiar with the standard format, and the man pages are read as input by other programs. The html version of the man pages is created by a Perl script, and `catman` creates index files by reading the man pages. These index files are searched by `man -k`, and are extremely useful.

In standard Unix usage, man pages are separated into a few general sections, and all the man pages for a particular section are kept in one directory. For instance, general user commands are section 1, and are kept in the subdirectory `man1`. Antelope generally follows this convention, using the following sections:

TABLE 1. Antelope Man Page sections (Release 4.5 and later)

Section	Directory	Description
1	<code>man1</code>	User Commands
3	<code>man3</code>	C library functions
3f	<code>man3f</code>	fortran library functions
3p	<code>man3p</code>	Perl functions
3t	<code>man3t</code>	Tcl/Tk library procedures
5	<code>man5</code>	File formats
8	<code>man8</code>	System Administration Commands

The `man` command looks at an index file named *windex* located in the root `man` directory. When you add a new man page to the manual pages in `$(ANTELOPE)/man`, you must update this index by running

```
% catman -w -M $(ANTELOPE)/man
```

You may also find that your system does not already have *windex* files generated in standard locations like `/usr/man`, `/usr/openwin/man` and `/opt/SUNWsprow/man`. The *windex* files can be created in these locations as well, by running

```
% catman -w -M directory
```

Exercises

1. Find the source code for the pfecho man page. Compare it with the formatted version and the html version. What is the most useful section?
2. Write a man page for a local program which doesn't have one yet. Install it, update the windex file, and make sure you can find it using the man command.
3. What is your MANPATH environment variable set to? Try the command

```
% man -k not-there
```

If you get error messages like these:

```
% man -k not-there
```

```
/opt/local/man/windex: No such file or directory
```

```
/usr/local/man/windex: No such file or directory
```

```
/usr/local/teTeX/man/windex: No such file or directory
```

```
run
```

```
% catman -w -M .
```

in the offending directories (you may need root permission).

Time is an extremely important part of any real time data collection system. Antelope represents time in the same standard way adopted by Unix: an epoch time measured in seconds since January 1, 1970 00:00 at Greenwich. Antelope epoch times are represented as doubles instead of integers, so that resolution can be in microseconds.

An epoch time is different from UTC time. It assumes a constant Gregorian clock, with 86400 seconds per day and 365 days per year, except in leap years, which have 366. As you are probably aware, a UTC time has occasional “leap seconds” inserted: some UTC minutes have 61 seconds instead of just 60.

This distinction between UTC and *epoch* time is generally inconsequential. Data loggers typically derive their time from a GPS clock which provides UTC time in the form of days and seconds. This is soon converted to an epoch time using the constant Gregorian clock. Because the epoch time ignores leap seconds, leap second data could show up in a waveform database as duplicate measurements for that particular second. To process data which fell across the leap second would require some special processing. Most of the time, however, the leap seconds can be safely ignored.

Epoch times are useful for calculations involving time. However, people are not typically comfortable with nine digit numbers representing times, and are more familiar with standard representations like “July 20, 1992 6:20 pm”. In addition,

people typically prefer to use their own time zone where at 12:00 noon the sun is about overhead. However, when comparing measurements made at widely different points on the earth, it's important to use a single time zone.

Consequently, Antelope includes facilities for translating between epoch times expressed in seconds and epoch times broken out into minutes, hours, months, days, and years. It also provides for some time zone conversion. Most programs allow input time strings (i.e., from the command line or a parameter file) to take a variety of different representations. However, some input formats can be misunderstood, and some nonsensical input is accepted. Most people become familiar with one of the many formats and stick with it.

The command line program `epoch` is the most accessible way to experiment with these different representations. `epoch` has an interactive mode where it reads input time strings, converts them internally into epoch seconds, and then prints out the epoch time in a variety of formats (like the standard Unix command `date`, you can also specify the output format). Here are some examples using this interactive mode of `epoch`:

```
% epoch
7/20/92 18:20 US/Mountain
  711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
21 jul 1992 0:20
  711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
1992 July 21 :20
  711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
00:20 1992-203
  711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
1992:203:00:20
epoch: Hour = 203 is too large
epoch: Doy = 992 is too large
epoch: couldn't convert input time
  1064660420.000 (270) 2003-09-27 11:00:20.000 UTC Saturday
1992:203:00:20:00
  711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
92/7/21 0:20
  711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
1992-7-21 0:20
  -21.000 (365) 1969-12-31 23:59:39.000 UTC Wednesday
1992-07-21 0:20
```

```

711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
1992:july:21:0:20:00
epoch: str2epoch: can't interpret ':'
epoch: couldn't convert input time
709950000.000 (183) 1992-07-01 00:20:00.000 UTC Wednesday
1992:jul:21:00:20:00
711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday

```

Unix includes an ability to name time zones, and deal with them to some extent. However, the Unix time zone naming convention appears to be non-standard: it differs from Solaris to Linux. Furthermore, the abbreviations which you may be familiar with, like *PDT* and *EST* are not easily mapped back to Unix names like *US/Pacific* and *US/Eastern*. `epoch` understands the Unix names as input, but prints the abbreviation on output.

```

% epoch 7/20/92 18:20 US/Mountain
711678000.000 (203) 1992-07-21 00:20:00.000 UTC Tuesday
% epoch -o US/Alaska 7/20/92 18:20 US/Mountain
711678000.000 (202) 1992-07-20 16:20:00.000 AKDT Monday

```

If only a time (no date) is entered, `epoch` assumes the date is January 1, 1970, and returns (among other things) the number of seconds corresponding to the input time:

```

2:13:35
8015.000 (001) 1970-01-01 02:13:35.000 UTC Thursday

```

A useful shorthand for the current time is *now*, and the command `epoch now` is a convenient method for obtaining the current UTC time. Another shorthand, useful primarily in the real time system, is a negative time, for example: *-0:12*, which is understood as 12 minutes before now:

```

now
960679934.597 (162) 2000-06-10 23:32:14.597 UTC Saturday
-0:12
960679220.717 (162) 2000-06-10 23:20:20.717 UTC Saturday

```

Look at the `epoch` man page or one of the reference guides for other examples, or experiment on your own.

Exercises

- 1) What does `+1:00` represent in an input time (to epoch, for example)?
- 2) What is your local time zone name, and where is it found on your system? (Hint: look for a directory named `zoneinfo`).

Parameter files pervade Antelope; they are used to configure programs which have more than a few parameters, to configure some libraries, and as a convenient way of passing parameters between programs on the orbserver. It is important to understand how to use them.

In general terms, they provide a flexible but standard way of specifying program parameters by name. Most Unix programs, and a few Antelope programs, take all their specifications from the command line. However, this quickly becomes cumbersome as the list of configurable parameters increases.

Parameter files associate a name with a value. For example, a line in the `dbevents` parameter file specifies the number of channels to display in `dbpick`:

```
channels          33  # channels to display in dbpick
```

Here the name is *channels* and the value is `33`. (The remainder of the line is a comment). This is the simplest type of value, a simple string. However, a value may take two other forms: an ordered list of values, or an array of name/value pairs.

Arrays and Lists

An ordered list is introduced with `&Tbl {` and ends with `}`. An example is

```
ordered      &Tbl{  
  first  
  second  
  third  
}
```

An array of name value pairs is introduced by `&Arr{` and ends with `}`. An example is:

```
capitals      &Arr{  
  Colorado    Denver  
  Alaska      Anchorage  
  Alaska      Juneau  
}
```

Here the value of `capitals{Colorado}` is `Denver`, and similarly the value of `capitals{Alaska}` is `Juneau`. Notice that the order of the name/value pairs in the array doesn't matter, unless a name is duplicated. Because there are two lines which begin with `Alaska`, the line that occurs later is used. This situation is not tagged as an error, though it might be puzzling in a more complex example.

Each item in an ordered list or value in an array may again be one of a simple string, or an ordered list, or an array of name/value pairs. There is no limit on the depth of the nesting, but typically two or three levels are adequate.

PFPATH

The standard parameter files are kept in the directory `$ANTELOPE/data/pf`. However, the `pf` routines usually look in the current directory for a parameter file after reading the file from `$ANTELOPE/data/pf`. If any name/value pair occurs in both files, the pair from the parameter file read last is used. Thus a local parameter file overrides the default values in the central directory.

Actually, the `pf` routines check for a parameter file in every directory named in the environment variable `PFPATH`. When no `PFPATH` environment variable is set, then only the two directories `$ANTELOPE/data/pf` and the current directory. are searched.

pfecho

pfecho is a useful tool for clearing up some puzzles like this, as well as for experimenting with parameter files. `pfecho -w orbserver` shows what parameter files are read for orbserver:

```
% pfecho -w orbserver
/opt/antelope/4.4/data/pf/orbserver.pf
orbserver.pf
```

pfecho can also find the value for a particular parameter:

```
% pfecho orbserver ringsize
ringsize:
    10M
```

The `-i` option causes pfecho to read names from stdin and print the corresponding values:

```
% pfecho -i orbserver
ringsize
10M
maximum_srcid
1000
^D
```

Extended Syntax

Often, elements in a parameter file are referred to by a simple name, as shown above. However, an extended syntax allows descending many levels into arrays and lists in a single step. The values of an array (indexed by name) are referenced by enclosing the name in curly brackets. The elements of an ordered list are referenced by an integer index; the index is enclosed in square brackets, and starts at zero.

Consider the following (modified) fragment taken from `dbdetect.pf`:

```
bands    &Tbl{
          &Arr{
                sta_twin      5.0
                sta_tmin      5.0
          }
    }
```

```
        &Arr{
            sta_twin      2.0
            sta_tmin      2.0
        }
    }
```

The first array in the list *bands* would be *bands[0]*, and the value for *sta_twin* would be *bands[0]{sta_twin}*:

```
% pfecho -i dbdetect
bands[0]
&Arr{
    sta_tmin      5.0
    sta_twin      5.0
}
bands[0]{sta_twin}
5.0
```

Comments

Parameter files may include comments. Generally, anything following a pound sign (#) is a comment. However, a pound sign can be included in a name or value by preceding it with a backslash (\).

Function Values

There are a few special function values:

- `&ask(prompt)`
This causes the user to be asked for the corresponding value, using the specified string *prompt*.
- `&ref(otherpf, name)`
This causes the value to be looked up in a different parameter file *otherpf* under another *name*.
- `&glob(pattern)`
This returns a list of filenames which match the specified glob style (like sh or csh) *pattern*.
- `&exec(command)`

This returns the output of the *command*.

- `&yourfunction(anything)`

By adding a routine to *libuser.so*, you can cause the pf routines to call your function to evaluate something.

Occasionally, it's necessary to include some text verbatim as a value in a parameter file. The best way to accomplish this is to use the special *function* value *&Literal{* which ends with *}*. Text in between the curly brackets, including balanced curly brackets, is part of the value.

```
text      &Literal{
This is some literal text
associated with the name text.
}
```

Dynamic parameter files

Most programs read their parameter files once at the beginning of execution. They do not see later changes until the program is restarted. There are a few exceptions; `rtexec(1)` checks its parameter file every second, using the routine `pfupdate(3)`. This routine checks the times on each file which makes up a parameter space, and returns a flag and an updated pf object whenever something changes. The program must then adjust accordingly.

Summary

Parameter files provide a standard, flexible method for specifying configurable parameters to programs. They are generally easy to read and understand with a common commenting style. They provide a way of introducing structure into the input. Because of these advantages, programs with many configuration options should use parameter files to specify the parameters.

Exercises

1. Enter the command

```
% pfecho -i rtexec
```

and figure out the syntax to find the value for orbserver in the Run array.
2. Look at the parameter files which configure the trace library (`trdefaults.pf`), the elog facility (`elog.pf`), and the travel time library (`trvltm.pf`). These con-

figure certain behaviors of the entire Antelope system. How could you arrange to change the system behavior for yourself without affecting everyone else?

3. Write a parameter file with `&glob` and `&ask` functions, and then read those values using `pfecho -i`.
4. Suppose the directory `$ANTELOPE/data/pf` contains a parameter file `x.pf`:

```
a &Tbl{
  one 1
  two 2
}
```

and the current directory contains this `x.pf`:

```
a &Tbl{
  five 5
  six 6
}
```

What is the result from the command below?

```
% pfecho x a
```

Antelope depends on Datascope databases. Consequently, you need to have a basic understanding of relational databases and some familiarity with specific Datascope tools, particularly `dbe`, to be an effective Antelope programmer. This chapter briefly discusses certain standard relational database operations and how to accomplish them with Datascope. You may wish to start by reading the Datascope tutorial, found in the docs directory of your Antelope distribution.

A Datascope database is made up of a collection of plain files which have a common basename (the database name), and different suffixes corresponding to the table names. For instance, the *site* table in the *demo* database is the file *demo.site*. The format of the table files making up a database is specified in a separate schema file.

Opening a database

The first step is to open a database, using the `dbopen` call. It takes two arguments, the path to the database, and the permission, which is either “r” (read-only) or “r+” (read-write). `dbopen` returns a database pointer.

Database pointers

Datascope calls use a construct called a database pointer. A database pointer should not be confused with a pointer in languages like C or Pascal; it is simply a collection of four integers. These integers identify some portion of a database, ranging from a specific field in a record in a table, to a whole record or range of records, to a whole table, to a whole database. They are actually indexes to structures internal to Datascope. However, they are often manipulated directly by the user. Each integer has a specific scope. These are, in order:

- database number

The database number is the index corresponding to the order in which databases were opened. A typical application may only open one database, in which case database number would be zero.

- table number

Table number is the sequence number of the table in the schema file. Views and tables created during a session are assigned sequentially larger numbers.

- field number

Field number is the sequence number of the field in the table.

- record number

Record number is the sequence number of the record in the table.

All these numbers follow the C convention of starting at zero, rather than the FORTRAN convention of starting at one. For example, the first record in a table has record number 0.

dblookup

A database, a table, and a field in a database all have names, of course; for instance, the `calib` field in the `wfdisc` table in the demo database

```
/opt/antelope/data/db/demo/demo
```

When a database is open, there is a corresponding database pointer to this field/table/database: the routine `dblookup` maps the names to a database pointer. The exact calling sequence depends on the language, but the routine always takes five arguments:

- an initial database pointer

-
- a database name
 - a table name
 - a field name
 - a record name

Only the input database pointer, and either table name or field name are typically specified in practice. Any of the last four arguments can be null (either zero or an empty strings) and they'll be ignored. When successful, `dblookup` returns a database pointer which is as specific as its arguments; specify a table and field and the corresponding database pointer fields are filled in.

Often, a user directly manipulates the record number, to step through a table, for example. Similarly, the table and field numbers can be manipulated directly, to get the fields in a table, or the tables in a database.

Special index values

Database pointer indexes may also take on a few special values: `dbALL`, `dbSCRATCH`, `dbNULL`, or `dbINVALID`.

When the field number is `dbALL`, the database pointer refers to the entire record. Similarly, when the record number is `dbALL`, the database pointer refers to the entire table. (However, it is not valid to have field number refer to a field and record number be `dbALL`).

The record number may be `dbSCRATCH` or `dbNULL`, in which case the database pointer refers to two special records for a table; the *scratch* record, and the *null* record. You may assemble a new record in the scratch record before adding it to the table, and you may read null values for fields or records from the null record.

All these special values correspond to negative integers. In `c` and `fortran` you use them directly in the code because they're defined in header files. You may also use `dblookup` to look up these values; this is the best way to obtain their values in `Perl` and `Tcl`.

Reading and Writing Fields

Most often, one reads a series of fields from a single record. For instance, one might read *sta*, *chan*, *time*, *nsamp*, and *samprate* from a single record in a *wfdisc* table.

This is accomplished through a call to `dbgetv`, which requires a database pointer identifying a single record of a table and a list of field names. It returns the values for those fields; the details vary according to the language.

A corresponding call, `dbputv`, is used to write values to a record in a table. Similarly, it requires a database pointer identifying the target record, a list of field names, and a corresponding list of values. Again, the details vary according to the language.

Deleting Records

In Datascope, specific record numbers are used when reading or writing records: e.g., record #10. Actually deleting a record changes the record number of all subsequent records, and invalidates any views computed with this table. This is often inconvenient, and consequently records are often marked for later deletion using `dbmark`. Marking a record sets all of the fields to null values; a later crunch operation with `dbcrunch` deletes all records which are null.

There is also a `dbdelete` call, which acts immediately and is used only infrequently.

Subsets

The subset operation is pretty intuitive: find all records which satisfy some condition. The condition may be any Datascope expression which can be evaluated on the row. (See the man page `dbexpressions(5)` or the Datascope tutorial for detail about expressions). The `dbsubset` call requires a database pointer identifying a table, and an expression; it returns a database pointer to the new view, a list of all records which satisfy the expression.

Sorting

The sort operation orders the records according to a set of sort keys. The keys may be either plain fields from a record, or an expression which can be computed on the record. The `dbsort` call requires a database pointer identifying a table and a list of sort keys. It returns a database pointer to the new ordered view. There are some optional flags: you may reverse the sense of the sort, or cause the sort to keep only one record with a particular set of keys. Sorting `wfdisc` with the *unique* option on the keys *sta* and *chan* would generate a list of the unique station/channel pairs.

Grouping

A sorted table may be grouped using `dbgroup`; this means that consecutive records which are identical in some set of fields are separated into bins. Like `dbsort`, `dbgroup` requires an input database pointer identifying the input table, and a list of keys; the result is a database pointer to a new view. A grouped view is convenient when you want to perform some operation on a collection of records. For instance, a bulletin may need to print out all the arrival records associated with an origin, and so it might use a view which groups arrivals by their associated origin.

Joining Tables

With your data in a relational database, you must regularly look up data related to a record from one table in a record of a different table. The *join* operation combines records from two tables allowing you to get all the information from a single virtual record.

For instance, in the demo database, given data recorded at station AAK and referenced in the *wfdisc* table, you must look into the *site* table to find out the location of AAK, and into *sitechan* to find out the orientation of the sensors, and into *sensor* and *instrument* to find out the response of the instrument. The most convenient way to do this is to construct a join of *wfdisc*, *sensor*, *instrument*, *site* and *sitechan*. Each record in this new joined view would have a record from *wfdisc*, and matching records from *wfdisc*, *sensor*, *instrument*, *site* and *sitechan*. From each such record, you could obtain the reference to the waveform data, the latitude and longitude of the station, the orientation of the sensor, and a reference to the response file.

The `dbjoin` call is used to form such views, two tables at a time. For the example above, we would call `dbjoin` on *wfdisc* and *sensor*, then on the result and *sensor*, and so on. (Actually, it would generally be much faster to compute the joined view of *sensor*, *instrument*, *site* and *sitechan*, and then join *wfdisc* to that, because *wfdisc* typically has many more records than the other tables).

Natural joins

Joins are not necessarily simple. The operation of a join between two tables can be conceptualized as follows:

1. Find all combinations of a row from each table.
2. Keep only the combinations which satisfy some condition.

What is the join condition? Typically, it has something to do with the keys of the two tables. I.e., to join `wfdisc` with `site`, find the row in `wfdisc` where the station name matches, and the time range in the `wfdisc` table matches the time range in the `site` table. This is a fairly complex expression, as you'll discover if you try to write it out.

Normally, Datascope is able to infer the keys which join two tables, by inspection of the table keys specified in the schema. This may not always work (or may give an unexpected result). In this case, there are two alternatives: specifying the join keys explicitly to `dbjoin`, or specifying an explicit join condition to `dbtheta`.

Usually, the join condition consists of matching some set of keys from the first table with a corresponding set of keys in the second table. For the `wfdisc-site` join mentioned above, the join keys for `wfdisc` are `sta` and `time::endtime`. For `site`, the join keys are `sta` and `ondate::offdate`. You could explicitly provide these keys to `dbjoin`, but Datascope figures out these join keys without your assistance. You may encounter other situations where you must specify the join keys to get the right result, however.

Complex join conditions (Unnatural joins)

In some cases, the join condition is not a simple matter of matching keys. For instance, consider a join of `origin` and `site` for origins that are within 2 degrees of a station. The join condition might be:

$$\text{distance}(\text{site.lat}, \text{site.lon}, \text{origin.lat}, \text{origin.lon}) < 2$$

This join can't be computed with `dbjoin`; you must use `dbtheta` instead. It actually takes the conceptual approach mentioned above, of computing the expression for each combination of rows, and keeping combinations which satisfy the condition. `dbjoin`, in contrast, forms an index on the second table, and looks for matching rows using that index.

What is a view?

In Datascope, a view should be visualized as a matrix, where each row is a record. Each column corresponds usually to a database pointer; this database pointer identifies a single record in a base table. The operations described above take database pointers to either a base table or a view, and return a database pointer to a view.

When you look up a field in a view, you find the first field with the same name in the view. I.e., if you look up *sta* in a join of *wfdisc* to *site*, you find *wfdisc.sta*. Of course, you could look up *site.sta*, and get the site copy of *sta* instead. When you read or write a field of a view, you actually read or write to some corresponding base table.

A view is a somewhat static snapshot into a database. It is a set of references to specific records in the database. Changes in the fields of existing rows of the database are reflected in the view, if you get the value of a field. However, the view does not change as new records are added to the base tables involved; the view doesn't get new records.

A program which needs a dynamic window into the database must either recompute the view frequently, or may find the *dbmatches* operation to be useful. *dbmatches* finds the collection of records in a table which would satisfy a certain join condition; it does not create a new joined view, it just returns the list of records. The user must index through those records; this is more lightweight and more convenient in many situations.

Other operations

dbnojoin is complementary to a join; it finds all the records which do not join with a second table. Such records often indicate a database problem of some sort.

dbsever removes one table from a view, keeping only rows which are unique in the output view. A related operation is *dbseparate*, which forms a view of all the records from a particular table which participate in the input view.

The *dbprocess* operation takes a list of operations (like *dbjoin*, *dbsubset*, *dbgroup*,...), and performs them in order, returning a view. The program *dbverify(1)* uses this mechanism to specify tests to perform on a database.

A view may also be taken apart with *dbunjoin*, which creates new tables containing only the records which participate in a view. A corresponding operation, *dbuntangle*, returns a list of all the records participating in the view for each table in the view.

Summary

Datascope database operations are the guts of the database system; it's largely these operations which provide the power of the database. Like arithmetic, there are not that many operations to learn: subset, sort, join, group and a few other closely related operations. All these operations are probably familiar from set theory in junior high school (or more recently, first grade). Relational database operations are based in set theory.

Exercises

1. Write out the join expression between the `wfdisc` and `site` tables.
2. Try out each of the following operations in `dbe`: `dbjoin`, `dbsubset`, `dbsort`, `dbnojoin`, `dbsever`, `dbseparate`, `dbtheta`.

One of the most difficult and neglected part of any program is handling errors which may occur. It's difficult to anticipate all the possible error conditions; novel uses and situations often expose important gaps in error handling.

Antelope can't help with the design which must be put into error detection and handling. However, through the *elog* facility, Antelope does provide a mechanism for logging or announcing errors, and for handling some of the typical blunders which can cause core dumps.

This *elog* facility is essential for the development of standard libraries, because it provides a method for returning detailed error messages without necessarily causing diarrhea on `stderr`. Instead, the error messages are placed on an internal error log. Library routines typically leave detailed messages in this log, and return some type of error code (often -1).

When the user detects an error return, s/he may choose to print the error log, or to eliminate the error messages and quietly try a different approach. A graphical application may choose to display the error messages in a dialog box, while a command line application might simply print them on `stderr`. The Antelope real time system typically puts messages into log files, but it could instead save errors into an orb-server.

The `elog` facility also attempts to intercept bus errors and segmentation faults. By default, it attempts to print a stack trace when one occurs. As an option, it can start a debugger like `dbx` or `ups`, allowing you to immediately investigate what went wrong.

There are a few basic ideas behind the `elog` facility: a stack of messages, an urgency code associated with each, and a delivery method. The logging levels from lowest to highest are *log*, *debug*, *notify*, *complain*, *die*, and *fault*. *log* level messages are generally not delivered immediately; instead they are placed onto a stack. Only library routines typically generate *log* messages. Applications may generate the other four levels as a deliberate part of error handling, while the *fault* level is detected by hardware, causing an interrupt which is intercepted by the `elog` routines.

Each level has an associated delivery list. When any message is delivered, the entire stack of log messages is also delivered. Messages are delivered according to the delivery list specified for that level. Each message may be delivered to multiple files. For instance, all *fault* and *die* messages might be put into one file, while corresponding *debug* messages might end up in a different file.

Each `elog` message has three components: a tag, the message, and (optionally) any error message corresponding to the setting of `errno`. The tag may be configured either via environment variables or with the `elog.pf` parameter file. The default tag includes the name of the program and a string indicating the severity, e.g. **debug**, **complain**, **fatal**, **alert** or **fault**. The tag may optionally contain the local time or epoch time, process id, thread id, or hostname. These options are convenient in the real time system, where errors are logged and may not be noticed until much later.

Summary

The error log facility provides a standard means of reporting errors. Using the `elog` facility should simplify your own development and debugging, help to ensure useful error output, and allow your own programs to integrate more fully with standard Antelope programs. Specific interfaces to this facility, which vary slightly from one language to another, are covered in later chapters and/or in various man pages.

Exercises

1. How would you make the `rtexec` time tags show local time rather than UTC? (This is probably a bad idea in practice).
2. In verbose mode, the real time detector program `orbdetect` prints extensive information about its calculations. Normally, this ends up in the `orbdetect` log file, but perhaps it would be more convenient if such informational messages were separate from actual error messages. Set up `elog.pf` to save debug messages in a different log file.
3. If you're a C or fortran programmer, set up an environment variable so that your favorite debugger is started when a program dies due to a segmentation fault or bus error.
4. Occasionally, it's useful to see the log messages as they occur, rather than later when the program chooses to print them, or not at all if the program is ignoring return codes. Set up an environment variable so these are printed immediately, and leave it set for a few days to see what messages you're missing.

Shell scripts are one of the innovations of the Unix environment, and a good starting point for novice users to begin programming. The basic idea is to put the commands you would type at the command line into a file and execute that file. Pipes (where the output of one program is fed to the input of another program) make this idea much more powerful. This leads to a typically Unix style: writing small programs which act as filters. These filters are then combined in fairly simple ways to accomplish some surprisingly complex tasks.

A large part of learning Unix is learning the syntax for pipes and i/o redirection, and then learning the many simple (and not so simple) filters Unix provides. This chapter does not attempt to repeat what has already been done very well elsewhere; rather, we look at a few examples of scripts which might be interesting to the Antelope operator.

Time Calculator

`epoch` is a standard utility provided by the Antelope package, as described in the previous chapter on time conversion. However, it's sometimes interesting to perform calculations with time -- i.e., how long between two dates, or what will the date be in *x* days? To answer questions like these, we'll make a simple script `tcalc` which uses `epoch`.

The basic idea behind the time calculator is to read time in a variety of formats and convert to an epoch time in seconds using `epoch`. Arithmetic is easy once the time is in seconds, so the additions and subtractions are straightforward using a standard Unix utility `expr`. Finally, `epoch` is used again to print the result in a more recognizable format.

Try this out by hand at the command line first, to calculate the number of days since April 15:

```
% epoch now
961088872.113 (167) 2000-06-15 17:07:52.113 UTC Thursday
% epoch April 15 2000
955756800.000 (106) 2000-04-15 00:00:00.000 UTC Saturday
% expr 961088872 - 955756800
5332072
% epoch 5332072
5332072.000 (062) 1970-03-03 17:07:52.000 UTC Tuesday
```

There are 62 days between June 15 and April 15. Typing in the commands like this may quickly uncover problems in the strategy. In this case, for instance, one soon discovers that `expr` handles integers only.

To make a program out of these commands, you can just put all these statements into a file named `tcalc` and run the command `chmod +x tcalc`. Then whenever you need to know the time difference between April 15 and June 15, 2000, run `tcalc`. Obviously, this particular sequence of commands is not much good as a script by itself. However, some useful scripts are almost this simple -- see the bulletin script below, for example.

To make this script more useful, we need to read arguments from the command line and use shell variables to hold intermediate results like the time in seconds.

It's generally good practice to begin a script with a line like the following:

```
#!/bin/sh
```

This tells Unix that this script should be interpreted by `/bin/sh`; Unix arranges to execute `/bin/sh` with its `stdin` pointing to the script. `/bin/sh` ignores this first line because `#` indicates a comment line to it.

As a debugging tool, it's often interesting to add a `-x` or `-v` option following the `/bin/sh` in the first line. Either causes each line of the script to be printed as it is executed.

Many people use another shell (like `/bin/csh` or `/bin/tcsh`) for interactive use, usually because of special features useful at the command line. It's possible to write scripts with `/bin/csh` or any other shell, but `/bin/sh` is a good choice. Later chapters use Tcl/Tk and Perl as scripting languages.

Here's a slightly more useful version of `tcalc`, using 3 arguments from the command line and shell variables:

```
#!/bin/sh -x
if [ $# -ne 3 ] ; then
    echo Usage: tcalc time1 {+|-} time2
    exit 1
fi
time1=`epoch +%E "$1" | sed 's/...//`
time2=`epoch +%E "$3" | sed 's/...//`
result=`expr $time1 $2 $time2`
epoch $result
```

Shell variables are introduced by the `$` (dollar sign). `##` is the number of arguments; if the script gets the wrong number of arguments, it prints an error message. Otherwise, it passes argument `$1` and `$3` to `epoch` to convert to seconds. `sed` is used to convert the floating point output of `epoch` to an integer. `expr` is then used to add or subtract the seconds, and `epoch` is used to print the results.

A Seismic Bulletin

Antelope software runs on a few relational databases. A wide variety of information is kept in these databases. One database contains the actual waveform data and event information (picks and origins). Another has information about processes run and resources used. Useful reports from these databases can easily be generated by combining a few Datascope database commands.

Here is an example of generating a day bulletin from the real time event database. This database is typically kept in the `db` directory where the real time system is run. It accumulates waveform data and automatic picks and locations. Usually, these automatic picks and locations are reviewed by an analyst (using the program `dbloc2`). After review, an event bulletin might be generated.

The basic idea here is to

- subset the origin table to contain only events of interest
- join the origin table with the event table, the assoc and arrival tables
- sort to put the events in order by event id
- display the parameters of interest.

Once again, trying this out by hand at the command line is a useful exercise. To play with this, your own live database would be convenient; otherwise try using

```
/opt/antelope/data/db/demo/demo
```

With a live database, you might choose to subset out only origins occurring in the previous 24 hours, with a command like:

```
% dbsubset db/az.origin 'time > _-24:00_' | \  
dbselect - orid 'strtime(time)' \  
         'distance(lat,lon,33.6117,-116.4594)' auth  
90657 6/14/2000 22:26:42.000 1.0668 CU  
90654 6/14/2000 22:26:40.776 1.0754 UC:flv  
90660 6/15/2000 11:10:49.000 89.771 QED  
90675 6/15/2000 18:55:31.763 0.94405 orbassoc mag
```

Here we subset the origin table to get only origins from the last 24 hours, and show the origin id, the time of the event, the distance from a station (*PFO*), and the event author. Two of these origins come from different catalogs (*CU* and *QED*), one is reviewed by *UC:flv*, and the last is an automatic location from orbassoc. Try running `dbhelp css3.0` to learn the contents of the origin table (and assoc, arrival and event), to see what else might be displayed.

A bulletin might show several origins corresponding to different estimates of the epicenter for the same event, and could include a list of arrivals. Given some input view, a dbjoin, dbsort, and a dbselect with more options accomplish this:

```
dbjoin - assoc arrival event | \  
dbsort - evid origin.time arrival.time | \  
dbselect - '" "' arid sta phase \  
          'strtime(arrival.time)' timeres \  
          -group evid orid time lat lon depth auth \  
          -pre orid auth lat lon depth 'strtime(time)' \  
          -group evid -pre '"Event #' . evid '
```

Part of the results might look like:

```
Event #30751
90654 UC:flv 32.9176 -115.4770 21.5863 6/14/2000 22:26:40.776
    375384 SMTC P 6/14/2000 22:26:47.300 0.152
    376334 SMTC S 6/14/2000 22:26:52.397 0.622
    .
    .
90657 CU 32.8900 -115.5200 0.8000 6/14/2000 22:26:42.000
    375384 SMTC Pg 6/14/2000 22:26:47.300 0.500
    376334 SMTC Sg 6/14/2000 22:26:52.397 2.039
    .
    .
```

If your security rules permit it, a script something like this might be connected to the finger daemon, to implement a finger server for your own catalog. You might want to show different parameters, or select only events which meet certain criteria.

An email alert

Network operators are often interested in being notified when certain things happen. The arrival of a new seismic event is a typical example, but there are myriad other possible triggers: disks filled up, station not reporting, or a system breakin. We will look at the beginnings of a script to report new seismic events.

This script works by reading records from the end of the origin table, performing some screening, and sending email if the origin satisfies the right criteria. This script introduces some other interesting features:

- shell subroutines
- the “here” document used to layout the email
- using the shell’s simple parsing capabilities to separate origin records into fields
- dbcalc used to perform a distance calculation

Here’s the script:

```
#!/bin/sh
DB=db/anza
PEOPLE=someone@somewhere.net
REFLAT=33.6117
REFLON=-116.4594
```

```
REFSTA=PFO
RECENT=600 # only events occurring less than $RECENT seconds
ago make email

email() {
    distance=`echo "distance($REFLAT,$REFLON,lat,lon)" |
        dbcalc $DB.origin orid==$orid`
    epoch=`epoch +%H:%M %a %D %Z" $time`
    local=`epoch -o $TZ +%H:%M %a %D %Z" $time`
    /usr/bin/mailx -s "($lat, $lon) @$local" $PEOPLE <<EOF
A new event has occurred at
    lat    = $lat
    lon    = $lon
    depth  = $depth
    time   = $epoch
    local time = $local
    Distance from $REFSTA = $distance
EOF
}

screen() {
    now=`epoch +%E now | sed 's/'...//`
    tevent=`echo $time | sed 's/'.....//`
    tdelta=`expr $now - $tevent`
    if [ $tdelta -le $RECENT ] ; then
        email
    fi
}

read_origins() {
while true
do
    read lat lon depth time orid evid jdate nass ndef ndp grn srn
    etype junk
    screen
done
}
```

```
tail -f $DB.origin | read_origins
```

At the beginning, various variables are set, specifying the database to look at, and various other criteria. After that, the script is most easily read from the bottom.

1. First, `tail` is used to read continuously from the end of the origin table, taking advantage of the fact that Datascope data records are simply lines in a file. These lines are then fed to the subroutine `read_origins`.
2. `read_origins` separates the record into the various leading fields like latitude, longitude and time, using the simple parsing based on white space which is built into the shell. The values are left in the corresponding shell variables.
3. `screen` calculates how recent the current event was. Only events which have happened very recently are reported -- this partially repairs the defect in using `tail` to watch for new records in the origin table. When it starts, `tail` always gets the last few records, but email-alert shouldn't send out email for old origins. This line might also effectively screen out events being imported from remote catalogs (presuming the remote catalog is relatively out of date by the time it's received). Other tests could be applied at this point also.
4. When the event is recent enough, email is called. `dbcalc` is used to calculate the distance from the event to a reference station. Then the body of the email is composed using a *here* document. Everything between `<<EOF` and the line beginning `EOF` is fed as input to `mailx`. However, parameter substitutions are performed on this text first.

This script is rather rudimentary, and has a number of flaws:

- reading directly from the origin table with `tail` does not work after the origin table is truncated. Normally, the real time system is set up to clean old records out of the database after some period, making tables like origin suddenly shorter. The script would need to be restarted after each such cleanup.
- The screening criteria is too simple.
- Specifying the database, email recipients and the reference station with explicit statements inside the script is poor practice. It would be better to get this data from the command line.

Nevertheless, this script demonstrates an approach to implementing an alert mechanism. Some cell phone and/or beeper providers have a mechanism for sending a small email message to the phone or beeper. However, this varies significantly from one site to another, as do the criteria and contents of an alert. That's why creating these alerts is a custom development effort for a network.

predict

This is a simple example which uses a number of Antelope interfaces: parameter files, epoch time calculations, database access, and travel time calculations. Given coordinates for an event, it shows predicted arrival times for the p-arrival at the stations from a site table, in the order of arrival.

```
#!/bin/sh
lat=$1
lon=$2
depth=$3
time=$4
epoch=`epoch +%E $time`
echo "time='$time'"
echo "epoch='$epoch'"
table=`pfecho -q predict table`
dbsort $table "distance(lat,lon,$lat,$lon)" |
    dbselect - staname "strtime($epoch+ \
        ptime(distance(lat,lon,$lat,$lon),$depth))"
```

The same problem is implemented in later chapters in other languages. Utility programs like `epoch` and `pfecho` make it possible to implement this as a shell script; indeed, it's a bit more compact than the other implementations. On the other hand, there's no compact way to print out the predicted local time in addition to the epoch time.

Summary

Shell scripts are easy to create, and an essential tool for a Unix programmer. They require some familiarity with a variety of Unix and Antelope tools, but this familiarity also increases proficiency at the command line. The scripts in this chapter used `epoch`, `dbcalc` and assorted database command line programs (`dbjoin`, `dbsort`, `dbsubset`, `dbselect`). There are, of course, man pages for these programs with a lot more detail about their operation. Some other programs which might be useful are `pfecho(1)` and `trsample(1)`.

On the other hand, shell scripts have some serious limitations. It's difficult to perform calculations and manipulate strings. Error handling is usually poor, and in general, one quickly runs into difficulties when attempting to solve more complex problems. The following chapters explore other scripting languages which surmount these difficulties and provide new capabilities.

Exercises

1. Modify `tcalc` to print a date when the result is bigger than a year, and a number of days and hours when the result is less than a year.
2. Modify `tcalc` to have an interactive mode.
3. Use `trsample` to calculate statistics on waveforms near P arrivals.
4. Modify `email-alert` to fix some of the flaws described above.

Tcl/Tk is an interactive programming environment originated by John Ousterhout. Most programs with a graphical user interface in Antelope are programmed with a Tcl/Tk script. A few examples are `dbevents`, `dbe`, `qtmon` and `displaytgrid`.

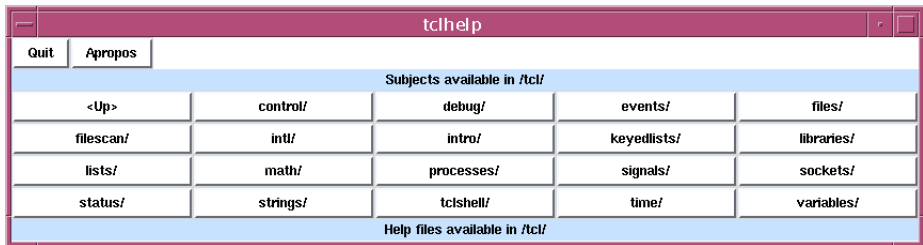
Tcl is a simple language which is similar to `csh`, but can be used for more complex tasks. It has a compact notation for associative arrays, but an ironically clumsy syntax for lists. Like `csh`, the quoting conventions are a weak point.

Tk is a set of graphical widgets initially written in conjunction with Tcl. It is an elegantly concise widget set, and is by far the simplest method for creating GUI's. Consequently, it has been ported to other languages, including Perl and Python.

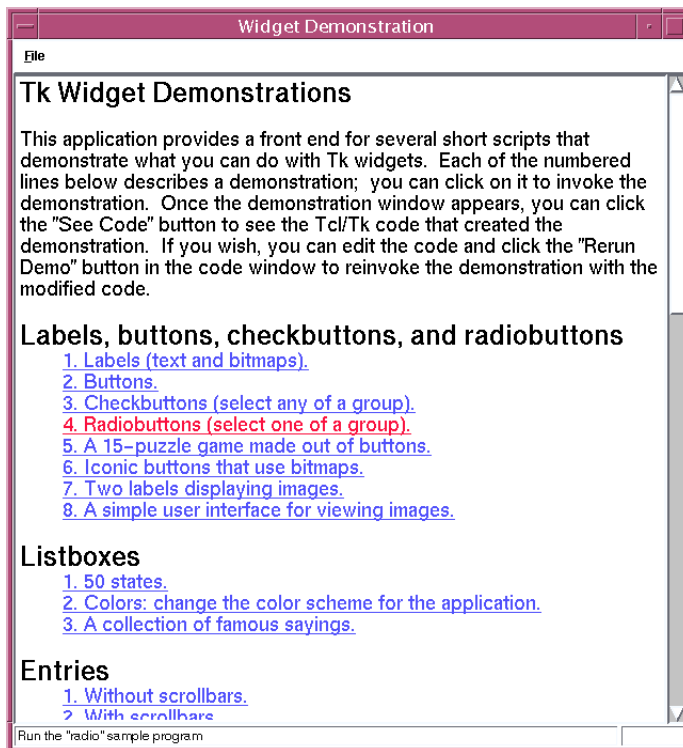
Learning Tcl/Tk

To learn about Tcl/Tk, refer to the books in the References; John Ousterhout himself wrote one introduction. There are also a number of programs which you may find useful, both as you're learning and later.

For instance, you will probably find the utility `tclhelp` to be very helpful. It is a graphical interface to the Tcl help facility, and provides a quick reference for Tcl and Tk commands. There is also a very complete set of man pages for questions not answered by `tclhelp`.



When building a GUI, it's oftentimes extremely useful to see a simple example. The `widget` demo provides exactly that service: simple examples of most of the widgets. It's probably the easiest way to explore the basic widget set.



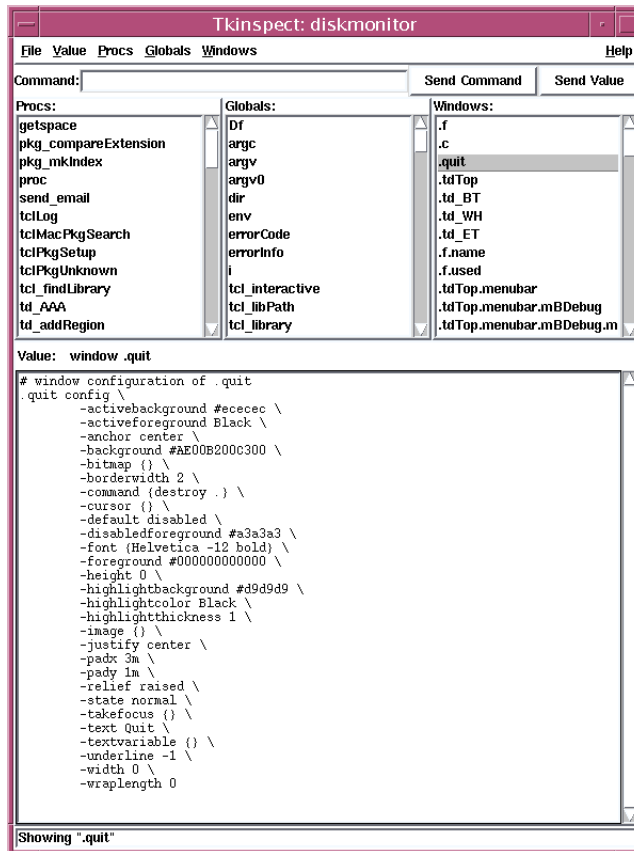
A unique and important strength of Tcl is that it is interactive. Often the quickest way to find out how something will work is to run `atcl` or `awish` and just type something in. This can be much faster and more rewarding even than reading the man pages.

Now, we've been a bit sloppy in terminology. Tcl is a *language*; `atcl` and `awish` are shells (like `/bin/sh` or `/bin/tcsh`) which read input and attempt to execute it as Tcl. The `awish` shell has the proper setup to create GUI's; otherwise it's the same as `atcl`. Tcl/Tk comes with similar shells named `tclsh` and `wish`; `atcl` and `awish` are the Antelope versions, almost identical except for some initialization. However, you'll want to use `atcl` and `awish` in your scripts, not `tclsh` or `wish`.

Development tools

When you have a Tk GUI, there are two additional programs which are very handy: `tkinspect` and `tkcon`. Both communicate with another application via the Tk `send` command. In each instance, you must select the other Tk application via a pull-down menu; under *File->Select Application* for `tkinspect`, and under *Console->Attach to->Interpreter* for `tkcon`.

`tkinspect` shows the widget hierarchy and how each widget is configured. Each Tk widget has many configuration options; getting everything configured just right is sometimes tricky. `tkinspect` lets you play with the configuration, without directly editing the source, until you get it right. Then you fix your source code.



`tkcon` is similar, but it provides a command line interface to a running application. This can be handy for figuring out what is happening internally without adding print statements to the code. You can type in a command and immediately see the result; you can query for the values of variables. You can even modify subroutines on the fly. This can be a very productive way to develop an application.

```

tkcon 2.1 Main
File Console Edit Interp Prefs History Help
>diskmonitor< (danq) 53 % array names Df
>diskmonitor< (danq) 54 % set Df(min)
10000
>diskmonitor< (danq) 55 % set Df(avail/)
212264
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %
>diskmonitor< (danq) 56 %

```

Finally, there is a graphical debugger which you may find useful. `tdebug` is a little less reliable for recent releases, but still potentially very useful. Like `tkinspect` and `tkcon`, you start it after you've run your program; then a pull-down menu is used to select your program. `tdebug` figures out what all the subroutines in the program are, and you can set a breakpoint at any of them. Then do something to trigger the breakpoint and single step through the code.

```

TDebug for diskmonitor
Debugger Options Selection Variables Help
Proc : [getspace (dir)]
global Df
after 1000 getspace $dir
set df [exec df -k $dir]
# Filesystem      kbytes  used  avail capacity  Mou
# /dev/dsk/c0t1d0s0 19620408 17927762 1496442  93% /
set percent [lindex $df 11]
set Df(percent$dir) $percent
Result:
Eval:
Delay: 300 - +
Stop Next Slow Fast Nonstop Break

```

Examples

Antelope provides Tcl interfaces to many of the standard functions used throughout the system. For instance, there are methods for using parameter files, ways to manipulate the database, and methods for creating graphs and plots of waveforms. This chapter explores some of these interfaces, developing scripts as examples.

dock

dock is a very simple program to illustrate how easy it is to construct a simple GUI. It reads a list of names and commands from a parameter file, and creates a dock of buttons; pressing a button executes the corresponding command. The bulk of the script is:

```
set commands [pfgetlist @dock#Commands]
foreach line $commands {
    regsub {[\t ]+.*} $line {} name
    regsub {^[^\t ]+[\t ]+} $line {exec } cmd
    set b .b$name
    button $b \
        -text $name \
        -command "$cmd &"
    pack $b -side top -fill x -expand yes
}
```

The first line reads the list of names and commands from the parameter file `dock.pf`, into the Tcl list `commands`. The `pfget` notation in Tcl is bit aberrant; here `pfgetlist` gets a list named `Commands` from the parameter file `dock.pf`.

The `regsub` commands use regular expressions to separate the names from the command lines (an `exec` is prepended to the command lines). A button is created, labeled with the name, which causes the command to be executed when it's pressed. Finally, the button is packed into the application window using the geometry manager `pack`.

Almost all of this is standard Tcl/Tk usage, which you can learn about in any of the standard books about Tcl/Tk. However, the script does use the `Datascope` extension package to get the `pfgetlist` command. How did that get included?

The beginning of the script is omitted above; the first few lines of any Tcl/Tk script cause the appropriate command interpreter to be run. For windowed applications, this is a wish (**w**indowing **s**hell) -- `awish` in this case. For a plain Tcl script, the corresponding interpreter is `atcl`.

In the examples directory, this script is named `dock.xwish`. The executable script is created from `dock.xwish` by `make`. The command `make dock` creates the application by copying a standard header followed by `dock.xwish` to the file `dock`. The standard header is:

```
#!/bin/sh
# \
exec $ANTELOPE/bin/awish $0 "$@"
package require Datascope
```

The package statement brings in the standard Datascope extensions.



The example shown arose from the following parameter file:

```
Commands      &Tbl{
update        antelope_update
ecrontab      ecrontab
rtdemo        rtdemo
dbhelp        dbhelp
tkcon         tkcon
tkinspect     tkinspect
xterm         xterm -j -sb -s -sl 300 -geometry 80x25
xcpustate     xcpustate -disk -wait
}
```

diskmonitor

Another way to use Tcl/Tk is to provide a graphical user interface to a program which has only a command line. For instance, the Antelope program `ecrontab` provides a graphical method for editing cron tables. As a less complex example, consider the `df` command, which provides information about disk partition usage. With real time data always arriving, there is a continuous risk of filling a disk. `diskmonitor` displays graphically the current disk usage, and sends mail when the disk has less than some fixed amount of space.

The necessary components are pretty clear:

- a function which runs `df` repeatedly and display the results: `getspace` uses the `after` command to rerun itself every second.

```
proc getspace {dir} {
    global Df
    after 1000 getspace $dir
    set df [exec df -k $dir]
# Filesystem          kbytes   used   avail capacity Mounted on
# /dev/dsk/c0t1d0s0   19620408 17927762 1496442    93%
/export/d

    set percent [lindex $df 11]
    set Df(percent$dir) $percent
    regsub % $percent {} percent
    set w [wininfo width .c]
    set l [expr $percent*$w/100]
    .c coords bar 0 0 $l 0
    if { $percent > 95 } {
        .c itemconfig bar -fill red
    } else {
        .c itemconfig bar -fill blue
    }
    set avail [lindex $df 10]
    set Df(avail$dir) $avail
    if { $avail < $Df(min) } {
        set now [clock seconds]
        if { [expr $now-$Df(alarm$dir) > $Df(alarm_period)] } {
            set Df(alarm$dir) $now
            send_email $Df(email) $dir $avail
        }
    }
}
```

```
    }  
}
```

- a function that sends mail: *send_email* shows how to open a pipe to *rtmail* to send alarm mail.

```
proc send_email {email dir avail} {  
    set subject "\"space low on $dir\""  
    set cmd "rtmail -s $subject $email"  
    set mail [open |$cmd w+]  
    puts $mail "  
    **** Warning ****  
        On [exec uname -n], the partition containing $dir  
        has only $avail kbytes of space left.  
    "  
    close $mail  
    puts "sent email to $email"  
}
```

- some code to do the initial setup and interpret the command line arguments. A line on a canvas object is the scale, with a label to show the usage numerically.

```
set Df(email) someone@somewhere  
set Df(min) 10000  
set Df(alarm_period) 3600  
set dir /  
set Df(alarm$dir) 0  
set Df(percent$dir) 100  
frame .f  
label .f.name -text "Disk Usage: $dir"  
label .f.used -textvariable Df(percent$dir)  
pack .f.name -side left -anchor w -fill x  
pack .f.used -side right -anchor e  
canvas .c \  
    -height 25 \  
    -relief flat \  
    -borderwidth 0 \  
    -highlightthickness 1  
.c create line 0 0 10 0 -width 50 -tag bar  
button .quit -text Quit -command {destroy .}  
pack .f .c .quit -side top -fill x -expand yes -anchor center  
getspace $dir
```

```
wm resizable . yes no
```

This is a lot more complex than the `dock` example; the only Antelope component is the script `rtmail`, which has the advantage of always returning immediately. If mail is not configured properly, piping to `mailx` can cause the script to hang.



Note that many of the parameters are hard wired in the script; in a functional application, they would either be command line options or kept in a parameter file.

tclpredict

Here we re-implement in Tcl the simple example which, given coordinates for an event, shows predicted arrival times for the p-arrival at the stations from a site table, in the order of arrival.

```
#!/bin/sh
# \
exec $ANTELOPE/bin/atcl $0 "$@"
package require Datascope
package require Tclx
set lat [lindex $argv 0]
set lon [lindex $argv 1]
set depth [lindex $argv 2]
set time [lindex $argv 3]
set epoch [str2epoch $time]
set table [pfget predict table]
set db [dbopen_table $table r]
set db [dbsort $db distance(lat,lon,$lat,$lon)]
set n [dbquery $db dbRECORD_COUNT]
loop i 0 $n {
    set db [lreplace $db 3 3 $i]
    set pred [dbeval $db \
```

```
ptime(distance(lat,lon,$lat,$lon),$depth)]
set pred [expr $pred+$epoch]
set staname [lindex [dbgetv $db 0 $i staname] 0]
puts [format "%-55s %s (%s)" \
$staname [strtime $pred] \
[epoch2str $pred "%H:%M:%S.%s %z" ""]]
}
```

This is a simple example of using several Antelope interfaces: parameter files, epoch time calculations, database access, and travel time calculations. Compare it with the implementations in different languages.

tkdbpick

By this time, you have probably used `dbpick` to inspect waveforms, and perhaps pick arrivals. And you may have noticed that there are many commands you can type in to `dbpick`. For instance, you can set the default phase for picking, change the waveforms displayed or their order. Rather than typing in commands, it is also possible to send commands to `dbpick` from a Tcl/Tk application. `dbloc2` does this to display waveforms for a particular event or arrival.

`tkdbpick` runs `dbpick` and then allows one to change the channels displayed, and to set the default phase for picks with a few buttons on a small window.

To display z channels only (presuming that channel names follow the usual convention), one may type into the `dbpick` command window:

```
sc *:. *Z
```

Similarly, to see horizontal channels, one might use an expression like:

```
sc *:. * [EN]
```

and

```
sc *:. *
```

to display all channels once again. `tkdbpick` sends these commands or similar commands to change the pick phase to `dbpick` at the press of a button. The routines `change_wf` and `change_phase` perform this service.

```
proc change_wf {} {
```

```
        global Appname Wf
        send $Appname "sc *:$Wf"
    }
    proc change_phase {} {
        global Appname Phase
        send $Appname "ph $Phase"
    }
```

The currently selected value of phase or channel selection code is contained in the global variable *Phase* or *Wf*.

The radiobuttons change the value of the global variables when they're pressed, and also call either `change_phase` or `change_wf`.

```
radiobutton .p_phase -command change_phase \
    -variable Phase -value P -text P
radiobutton .s_phase -command change_phase \
    -variable Phase -value S -text S
radiobutton .lg_phase -command change_phase \
    -variable Phase -value Lg -text Lg

radiobutton .z_wf -command change_wf \
    -variable Wf -value {.*Z} -text Z
radiobutton .ne_wf -command change_wf \
    -variable Wf -value {.*[NE]} -text NE
radiobutton .all_wf -command change_wf \
    -variable Wf -value {*} -text ALL
```

Finally, a quit button is always a good idea, and all the widgets must be positioned on the window, using the `grid` geometry manager here.

```
button .quit \
    -text Quit \
    -command {send $Appname quit ; destroy .}

grid .p_phase -col 1 -row 1
grid .s_phase -col 2 -row 1
grid .lg_phase -col 3 -row 1

grid .z_wf -col 1 -row 2
```

```
grid .ne_wf -col 2 -row 2
grid .all_wf -col 3 -row 2
grid .quit -col 1 -row 3 -columnspan 3
```

There are always two parts to adding a widget to an application: creating the widget, and specifying how to put it in the application window. The `grid` geometry manager is particularly easy; you specify what cells in a grid on the window your widget occupies.

This script runs `dbpick`, primarily to provide the `-appname` argument so that `dbpick` knows to listen for Tk send commands, and `tkdbpick` knows what application to send them to.

```
set Appname dbpick
exec /usr/openwin/bin/xterm -geometry 80x24+0-0 -e dbpick \
    -nostarttalk -geom 1000x700 -appname $Appname $argv &
```

The concept is quite similar to `dock`, but uses radio buttons instead of plain buttons. To get buttons in a regular array, it uses the `grid` geometry manager instead of `pack`. Finally, the command exercised at a button press is `send` instead of `exec`.



tkreorder

`dbpick` normally places traces on the screen in alphabetical order, or when an origin is associated, in the order of predicted arrivals, with time adjusted to line up the predicted `p` arrivals. It may be convenient (for picking) to arrange traces in order of increasing distance from a station. `dbpick` does not provide this facility itself, but the `tkreorder` script adds the capability.

The script works by asking `dbpick` what stations are displayed, then sorting the stations by distance from the station which the user specifies, and finally commanding `dbpick` to display traces in that order. It actually asks `dbpick` for the list of stations whenever the menu button is depressed and uses the returned list to build the menu of stations. Of course, it requires station coordinates for this, which it gets from an input site table.

The routine `sort_by_distance` first calculates the distance to each station,

```
proc sort_by_distance {sta} {
    global Dbpick
    set lat0 $Dbpick(lat$sta)
    set lon0 $Dbpick(lon$sta)
    foreach i $Dbpick(stas) {
        if { ! [info exists Dbpick(lat$i)] } {
            after 10 \
                tk_messageBox \
                    -message "no site table entry for $i" \
                    -type ok
            set x .001
            while { [info exists station($x)] } {
                set x [expr $x+.001]
            }
            set station($x) $i
        } else {
            set lat1 $Dbpick(lat$i)
            set lon1 $Dbpick(lon$i)
            set x [lindex [dbdist $lat0 $lon0 $lat1 $lon1] 0]
            while { [info exists station($x)] } {
                set x [expr $x+.001]
            }
            set station($x) $i
        }
    }
}
```

and then creates an array of stations indexed by distance.

```
set distances [array names station]
set distances [lsort -real $distances]
```

Notice that a large part of the routine is devoted to handling cases where the station did not appear in the site table. Now it has a sorted list of the distances, but it needs to send `dbpick` an ordered list of channels. The first step is to ask `dbpick` for the channels available. Then it creates the `dbpick`-style ordered list of channels *order*:

```
set chans [send $Dbpick(appname) echo %stachans]
set order {}
foreach i $distances {
```

```
        foreach j $chans {
            if { [string match $station($i):* $j] } {
                lappend order $j
            }
        }
    }
}
```

To save time, the waveform display is turned off when the list of stations is sent to dbpick. Then the display is limited to 40 channels again, and finally the waveform display is turned back on:

```
    set cmd "sw off"
    append cmd " sc [join $order ,]"
    append cmd " cw 1 40 sw on"
    if { [catch {set stas [send $Dbpick(appname) $cmd]} error] } {
        tk_messageBox \
            -message "reorder command failed: $error" \
            -type ok
    }
}
```

The routine below is called when the menubutton fromStation is pressed; it asks dbpick what stations are available, and then sets up the drop down menu which you see. If you watch dbpick's type-in window, you can see the commands going back and forth.

```
proc getStations {} {
    global Dbpick
    if { [catch {send $Dbpick(appname) echo %stas} Dbpick(stas)] } {
    } {
        $Dbpick(byStation) delete 0 end
        $Dbpick(byStation) add command \
            -label {No Stations in Window} \
            -state disabled
    } else {
        $Dbpick(byStation) delete 0 end
        foreach sta $Dbpick(stas) {
            $Dbpick(byStation) add command \
                -label $sta \
                -command "sort_by_distance $sta"
        }
    }
}
```

```
    }
  }
}

if { $argv == "" } {
    puts stderr "Usage: $argv0 database"
    exit 1
}
set database [lindex $argv 0]
set Dbpick(appname) dbloc_dbpick
set Db [dbopen $database r]
```

The site table is loaded directly into a global array as `tkreorder` starts. Notice how this script uses one global array `Dbpick` to save all its global variables.

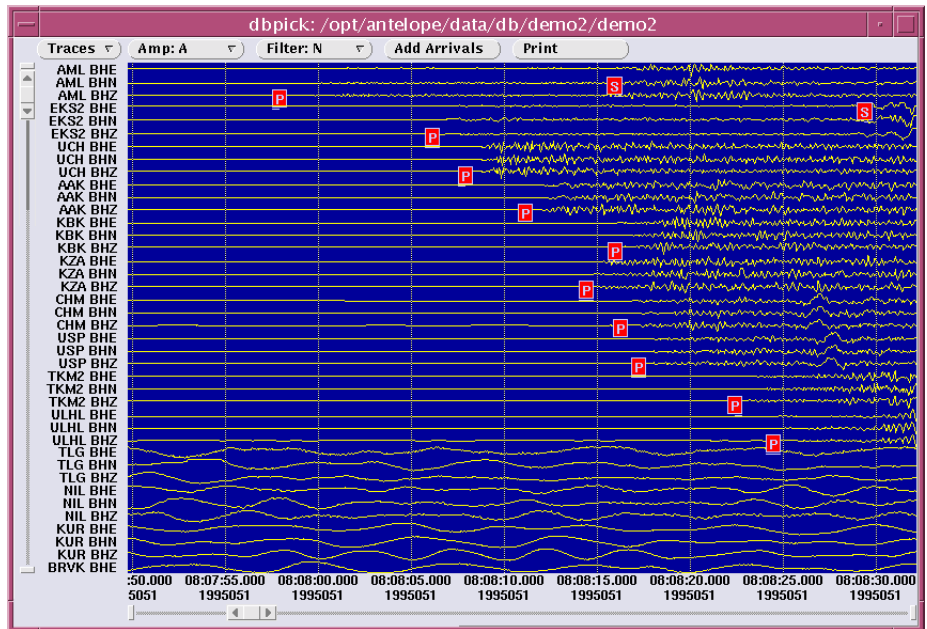
```
set dbsite [dblookup $Db 0 site 0 0]
set n [dbquery $dbsite dbRECORD_COUNT]
loop i 0 $n {
    set sta [dbgetv $dbsite 0 $i sta]
    set Dbpick(lat$sta) [dbgetv $dbsite 0 $i lat]
    set Dbpick(lon$sta) [dbgetv $dbsite 0 $i lon]
}
wm title . "tkreorder"
wm iconname . "tkreorder"
set mb .byStation
set Dbpick(byStation) [tk_optionMenu $mb Dbpick(sta) fromStation]
$Dbpick(byStation) configure -postcommand getStations
pack $mb
button .quit \
    -text Quit \
    -command "send $Dbpick(appname) quit ; destroy ."
pack .quit -fill x
```

The remainder of the script is setting up the menu button and quit button; the `pack` geometry manager is used here. It's a bit simpler than `grid`, and also more appropriate for windows which can be resized by the user. You might want to look at the `BLT table` geometry manager also.



Here's an example using tkreorder from the database

`/opt/antelope/data/db/demo2`



dbmap

This script is an example of using the MapWidget(3t) to annotate a simple interactive map. The map is very similar to the dbloc2 arrival map, and allows zooming in by selecting a region with the mouse. The selection can also be adjusted by dragging with the second mouse button. It provides several different projections, among other options.

The heart of the script is the routine `draw`, which takes the following arguments:

- center coordinates of the map
- range from the center.

`draw` then places circles at all the lat/lon locations in the specified input table. For the site or origin table, it also labels the circle with the station name or orid.

```
proc draw {latc lonc range} {
    global db database name field
    set stasz [expr $range/180. * 300000.]
    set textsz [expr 3*$stasz]
    g_textsize $textsz [expr $textsz/.6]
    g_polyfill 1
    set n [dbquery $db dbRECORD_COUNT]
    loop i 0 $n {
        set coords [dbgetv $db 0 $i lat lon]
        set trans [eval pj_fwd $coords]
        set lat [lindex $trans 0]
        set lon [lindex $trans 1]
        g_color red
        g_circle $lon $lat $stasz
        if {"$field" != ""} {
            set val [dbgetv $db 0 $i $field]
            g_drawstr " $val"
        }
    }
}
```

The remainder of the script just sets up the `draw` routine: first opening the database and setting up the label (if any):

```
set db [dbopen_database $argv r]
if { [lindex $db 1] < 0 } {
    set db [dblookup $db 0 site 0 0]
}

set database [dbquery $db dbDATABASE_NAME]
set db [dblookup $db 0 0 lat 0]
set name [dbquery $db dbFIELD_BASE_TABLE]
```

```
switch $name {
    site      {set field sta}
    origin    {set field orid}
    -        {set field {}}
}
```

Then it creates the `MapWidget` instance. `MapWidget` has numerous options, described in the man page.

```
wm title . "$database $name"
MapWidget .map \
    -center "0 0" \
    -range 180 \
    -auto_hires 5 \
    -coverage "fill coasts political" \
    -projection robin \
    -buttons "projection zoomout hires graticule save" \
    -indicator 1 \
    -label 1 \
    -quit 1 \
    -command draw

pack .map -side top -expand yes -fill both
wm geometry . 600x500
wm maxsize . 1000 1000
wm minsize . 100 100
```

`MapWidget` is itself a Tcl script, found in the directory:

```
$ANTELOPE/data/tcl/library
```

`dbmap` adds this directory to `auto_path`, so that `MapWidget.tcl` is automatically loaded when needed.

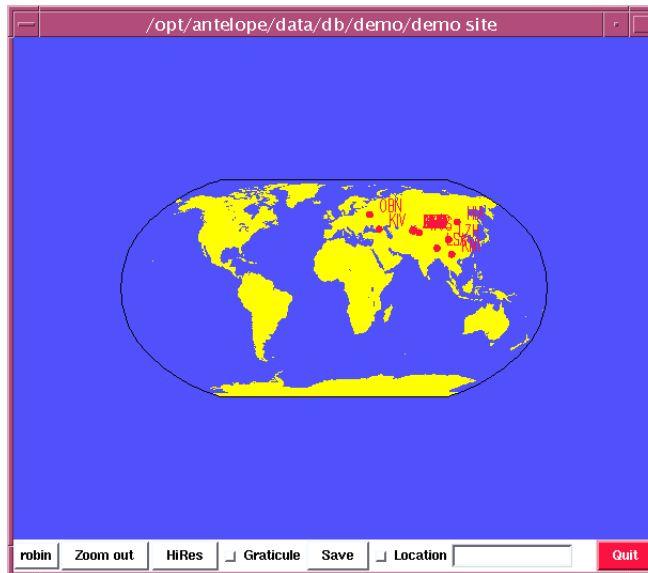
```
set auto_path \
    [linsert $auto_path 0 $env(ANTELOPE)/data/tcl/library]
```

In addition, the script requires the *Vogle* package (for the graphics library), the *Datascopes* package (for database operations) and the *Dbe* package for the map access and drawing routines.

```
package require Dbe
package require Datascope
package require Vogle
```

Finally, here are the results, for the command

```
% dbmap /opt/antelope/data/db/demo/demo
```



Summary

Tcl/Tk is a very quick and convenient way to write programs with a graphic interface. It is easy to learn, and adequately fast for many graphical applications. However, it is much slower than native c, for example, and was not intended for large applications. Many of these limitations can be avoided if you write your own Tcl extensions in c; Antelope provides many such extensions. Adding c extensions is straightforward, but you must look to the references for help and examples.

Exercises

1) Use `tkinspect` to look at the `dock` application as it's running, and adjust the color of the buttons and their relief.

- 2) Change `dock` to make the buttons line up horizontally instead of vertically. Make the dock always appear in the same location on the screen with a `wm` command.
- 3) Extend `diskmonitor` to monitor several disks at once.
- 4) Attach `dbmap` to `dbe` by editing `dbe`'s parameter file `.dbe.pf` and adding a new entry to the `Graphics` menu for tables which have `lat` and `lon`: site, origin and arrival.
- 5) Combine `tkreorder`, `tkdbpick`, `dock` and `dbmap` to customize `dbloc2` for your application. Make buttons to reorder or eliminate waveforms according to your rules, make specialized maps and add special notations to them, and so on.

Perl is a language originally designed by Larry Wall, and currently supported by a large user community across multiple architectures and operating systems, including most (if not every) flavor of Unix, Linux, Windows and MacOS. It is a favorite of system administrators, because it is fast, powerful, and probably the best general purpose tool for string manipulation available.

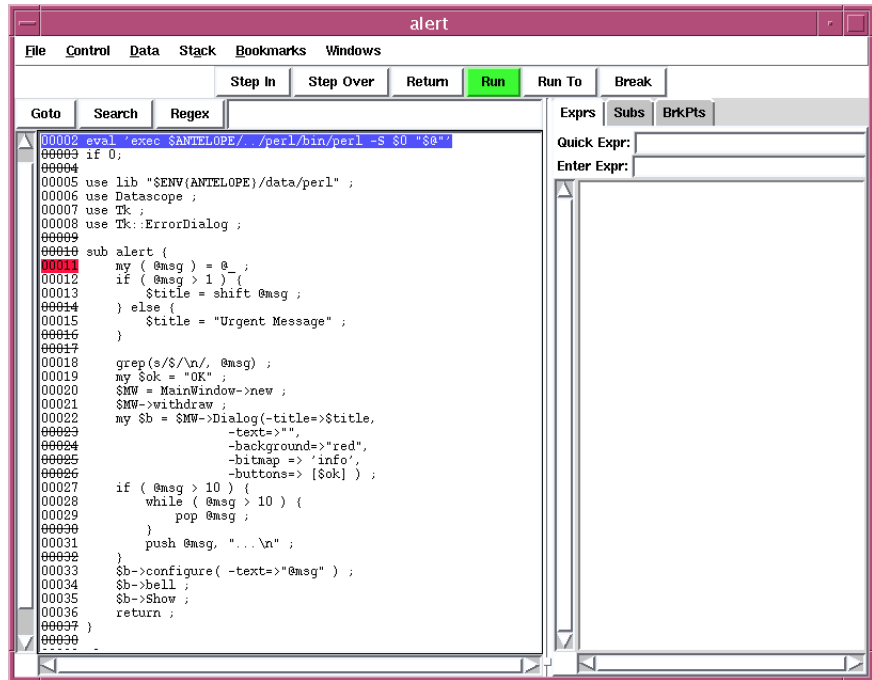
Many Antelope utilities are written in Perl: `rtexec`, `event_archive`, `ldlibs`, `rtsys`, `pktmon`, `autoDRM` and `rtdemo`, for example. Antelope extensions provide access to almost all the base Antelope library functionality: time conversion, error handling, path manipulation, parameter files, database operations and orbserver connections.

Learning Perl

Getting started in Perl can be a bit difficult, but there are many good books. *Programming Perl* is a classic, and you will probably want the *Perl 5 Pocket Reference* also. This chapter presumes familiarity with Perl, and introduces some of the functionality which Antelope adds to standard Perl.

There's an interesting visual debugger which you can run on most Perl scripts as follows:

```
% perl -d:ptkdb ./your-script arguments
```



It's largely self explanatory, and it may help you follow along what's happening in a script. It uses a port of the Tk widgets.

The Perl documentation is all in man pages, but you can also use a special Perl script `perldoc` to look up information. It's particularly useful for quickly finding the description of a function, like this:

```
% perldoc -f open
```

or for checking the Perl FAQ's for something:

```
% perldoc -q flush
```

There's a program for reformatting perl text named `perltidy`. You might find it useful for making someone else script readable, or for decoding obfuscated perl:

```
http://www.sysadminmag.com/tpj/obfuscated/
```

Examples

Which pf?

When you get a value from a parameter file object, either with `pfecho(1)` or from within a program, the underlying `pf` routines may actually combine multiple parameter files to get the result. It is not necessarily obvious what actual file contains the value returned. The script `whichpf` answers this question; you provide the parameter file name and the key, it prints out the file name.

The procedure is to use `pffiles(3)` to find the list of files which make up the parameter file object, and to then check each one in reverse order for the key in question. The first file containing the key is the defining file.

Here's the first step; get the list of files and reverse it. (You can see the results from `pffiles` with the command line program `pfwhich`).

```
@pffiles = pffiles($pf) ;
@pffiles = reverse (@pffiles ) ;

$ENV{'PFPATH'} = " " ;
```

The next line sets the `PFPATH` so that only the `license.pf` file is found, and then each file is checked in turn for the particular key:

```
foreach $i ( @pffiles ) {
    $i = ".$i" if $i !~ m"^\|^\.\/" ;
    $result = pfget($i, $key) ;
    if ( defined $result ) {
        print "$i\n" ;
        exit 0 ;
    }
}
```

The second line makes the filename explicit by prepending `./`, if it doesn't already begin with `/"` or `./`.

Use dbpick to make postscript

It's possible to create a postscript file from `dbpick`; the procedure is to setup the display as you wish, then enter the command `ps aplot.ps` in the type in window.

The minimal script `dbpick2ps` automates this procedure to a one line command, provided you know the time range and channels to display.

```
($database, $start, $stop) = @ARGV ;
$t0 = str2epoch($start) ;
$t1 = str2epoch($stop) ;
$t1 += $t0 if ( $t1 < $t0 ) ;
$sc = ($opt_s ? "-sc $opt_s" : "" ) ;
open ( DBPICK, "| dbpick -ts $t0 -te $t1 $sc -nostarttalk $data-
base " )
    || warn ( "running dbpick failed: $!\n" ) ;
print DBPICK <<"EOF" ;
ps plot.ps
quit
EOF
close DBPICK ;
```

Most of this script consists of reading the arguments from the command line. A standard `getopts` routine collects the options into variables named `$opt_X` where `X` is the option key. The three required arguments are collected with a single Perl statement that assigns the contents of a list on the right to a list of scalars (`$database, $start, stop`) on the left. The script allows the time range to be either a start time/end time, or a start time followed by a time period. It uses `str2epoch` to allow any reasonable input time format, which is converted to an epoch time using `str2epoch`.

Finally, `dbpick` is run such that it is reading from the pipe `DBPICK`. Down this pipe `dbpick2ps` prints the two necessary commands: `ps` and `quit`.

This still requires that the `DISPLAY` variable be set so that `dbpick` can paint to an X window. Suppose you wanted to run this in a `cron` job, or without a display screen? This could be accomplished by running a hidden display with `vncserver` (Virtual Network Computing). `vncserver` creates a virtual X display to which `dbpick` can write; the `cron` job must set `DISPLAY` to that display, and run `dbpick2ps`:

```
#!/bin/sh -x
DISPLAY=castle:1
ANTELOPE=/opt/antelope/4.4
PATH=$ANTELOPE/bin:$PATH
export DISPLAY ANTELOPE PATH
```

```
db=/opt/antelope/data/db/demo/demo
bin/dbpick2ps $db '5/17/92 21:55' '5/17/92 21:57'
```

As usual with cron jobs, it's also necessary to set up other environment variables like PATH properly. You may find `cronrun(1)` helpful in figuring out just what is missing before consigning your job to cron.

alert

It's sometimes convenient to bring up a window with a message. The `alert` script does just that. Perl also implements the Tk widget toolkit, used in this example. The script calls a subroutine with the command line arguments. The subroutine separates out the first argument as a title (if there's more than one argument), and puts the rest into the dialog window, truncating after 10 arguments. The window is drawn with a standard `Perl::Tk Dialog` widget, and has one button labeled ok.

```
use Datascope ;
use Tk ;
use Tk::ErrorDialog ;

sub alert {
    my ( @msg ) = @_ ;
    if ( @msg > 1 ) {
        $title = shift @msg ;
    } else {
        $title = "Urgent Message" ;
    }

    grep(s/$/\n/, @msg) ;
    my $ok = "OK" ;
    $MW = MainWindow->new ;
    $MW->withdraw ;
    my $b = $MW->Dialog(-title=>$title,
                      -text=>"",
                      -background=>"red",
                      -bitmap => 'info',
                      -buttons=> [$ok] ) ;

    if ( @msg > 10 ) {
        while ( @msg > 10 ) {
```

```
        pop @msg ;
    }
    push @msg, "...\\n" ;
}
$b->configure( -text=>"@msg" ) ;
$b->bell ;
$b->Show ;
return ;
}

alert @ARGV ;
```

Here's how it looks:

```
% alert "Time to go"
```



perlpredict

This is again the simple example which, given coordinates for an event, shows predicted arrival times for the p-arrival at the stations from a site table, in the order of arrival. It shows how several Antelope interfaces -- parameter files, epoch time calculations, database access, and travel time calculations -- appear in Perl.

```
: # use perl
eval 'exec $ANTELOPE/./perl/bin/perl -S $0 "$@" '
if 0;
use lib "$ENV{ANTELOPE}/data/perl" ;
use Datascope ;

($lat, $lon, $depth, $time) = @ARGV ;
$epoch = str2epoch ($time) ;
$table = pfget ("predict", "table") ;
@db = dopen_table($table, "r") ;
```

```
@db = dbsort( @db, "distance(lat,lon,$lat,$lon)" );
$n = dbquery (@db, "dbRECORD_COUNT");
for ( $db[3] = 0 ; $db[3] < $n ; $db[3]++ ) {
    $pred = dbex_eval (@db,
        "ptime(distance(lat,lon,$lat,$lon),$depth)" );
    $pred += $epoch ;
    ($staname) = dbgetv(@db, "staname" );
    printf "%-55s %s (%s)\n", $staname,
        strtime($pred),
        epoch2str($pred, "%H:%M:%S.%s %z", "" );
}
```

dbfilenames

Because of the database path (specified in a descriptor file), sometimes it's hard to keep track of where database tables reside. It could be in any of the directories along the path, and the actual file may have a different database name. The `dbfilenames` script makes the actual file explicit, as well as showing the table permissions and numbers of rows. It's also a simple example of using Perl to get information from a database.

The program simply opens the database named on command line, and uses `dbquery` to print out some general information about the database (like the schema). Then it checks each relation to see if there is a corresponding table with at least one row, and prints information from `dbquery` about that table.

```
require "getopts.pl" ;
if ( ! &Getopts('v') || @ARGV != 1 )
    { die ( "Usage: $0 [-v] database\n" ) ; }

use Datascope ;
$database = $ARGV[0] ;
@db = dbopen ( $database, "r+" ) ;
$db = dbquery (@db, dbDATABASE_NAME) ;
print "\ndatabase: $db\n" ;
$db = dbquery (@db, dbSCHEMA_NAME) ;
print "schema: $db\n" ;
$dbpath = dbquery (@db, dbDBPATH) ;
print "path: $dbpath\n" ;
$locks = dbquery ( @db, dbLOCKS) ;
```

```
print "locking: $locks\n" ;
$dbidserver = dbquery (@db, dbIDSERVER) ;
print "idserver: $dbidserver\n" ;

print "\n  relation      #rows w? add?  filename\n" ;
$n = dbquery ( @db, dbTABLE_COUNT ) ;
for ( $db[1]=0 ; $db[1]<$n ; $db[1]++ ) {
    $nrec = dbquery ( @db, "dbRECORD_COUNT" ) ;
    if ( $nrec > 0 || $opt_v ) {
        $table = dbquery ( @db, "dbTABLE_NAME" ) ;
        $file = dbquery ( @db, "dbTABLE_FILENAME" ) ;
        $writable = dbquery ( @db, "dbTABLE_IS_WRITABLE" ) ;
        $addable = dbquery ( @db, "dbTABLE_IS_ADDABLE" ) ;
        printf ( "%12s %8d %s %s %s\n", $table, $nrec,
            $writable ? "y" : "n",
            $addable ? " y" : " n",
            $file ) ;
    }
}
```

Here's some sample output:

```
% dbfilenames demo
database: demo
schema: css3.0
path:      ./{demo}
locking:  local
idserver:

      relation      #rows w? add?  filename
instrument      22 n   n demo.instrument
      sensor       39 n   n demo.sensor
      site         13 n   n demo.site
sitechan       39 n   n demo.sitechan
      wfdisc      18 n   n demo.wfdisc
```

dbemail

This script watches an origin table and sends email whenever a record is added to the table. It simply opens the table, checks to see how many records there are, and

then checks back every so often (every minute by default). Whenever a record has been added, it reads the record and uses mailx to send mail to the command line recipients.

This is, of course, entirely too simple for a real application. Probably it should check to make sure that it's a recent and nearby event, for instance.

```
require "getopts.pl" ;
if ( ! &Getopts('s:v') || @ARGV != 2 ) {
    die ( "Usage: $0 [-s sleep] [-v] database email\n" ) ;
}

use Datascope ;
$database = shift ;
$email = shift ;
$sleep = $opt_s ? $opt_s : 60 ;
@db = dbopen ( $database, "r" ) ;
@db = dblookup ( @db, 0, "origin", 0, 0 ) ;
$n = dbquery ( @db, "dbRECORD_COUNT" ) ;
print STDERR "starting with $n records\n" if $opt_v ;
for (;;) {
    sleep ( $sleep ) ;
    $n2 = dbquery ( @db, "dbRECORD_COUNT" ) ;
    if ( $n2 > $n ) {
        send_email ( $n, $n2, $email, @db ) ;
        $n = $n2 ;
    }
}

sub send_email {
    my ( $n, $n2, $email, @db ) = @_ ;
    open ( EMAIL, "|mailx -s 'new origins' $email" ) ;
    my ($lat, $lon, $depth, $time, $mb) ;
    for ( $db[3] = $n ; $db[3] < $n2 ; $db[3]++ ) {
        ($lat, $lon, $depth, $time, $mb) =
            dbgetv ( @db, qw(lat lon depth time mb)) ;
        printf EMAIL "%5.2f %8.3f %8.3f %6.2f %s %s\n",
            $mb, $lat, $lon, $depth, &strtime($time),
            &grname($lat, $lon) ;
    }
}
```

```
        printf STDERR "%5.2f    %8.3f %8.3f %6.2f    %s %s\n",
            $mb, $lat, $lon, $depth, &strtime($time),
            &grname($lat, $lon)
            if $opt_v ;
    }
    close EMAIL ;
}
```

NEIC_qed, NEIC_qed2

In the examples, you can find two scripts which automate reading web pages from the NEIC and adding new origins from those web pages to a local database. While we won't go into detail about these scripts here, they use Perl extensions for HTTP and HTML. Perl has a large, well organized library of routines which are generally freely available and implement a wide variety of special functions. Look at the web page:

<http://www.cpan.org/>

The outline of the programs is simple: figure out the last origin read, read and parse the web page, find any new origins, save them to the database, sleep for awhile, and try again. This is an application which uses some of Perl's string manipulation capabilities.

Summary

Perl is a language which is very powerful and has a wide audience and a strong crew of developers around the world. While the syntax can be tough to get used to, programs written in Perl are often shorter and more concise than equivalent programs in other languages. While it is not as fast as native C, it's usually faster than Tcl/Tk. Like Tcl/Tk, you can write extensions to Perl in c. The documentation is less clear for Perl: start with the `perlx(1)` man page. Alternatively, with the Inline extension (see CPAN), you can embed c directly into your Perl script.

Exercises

1. Change `dbpick2ps` to make it print out a record section plot for a particular event.
2. Unix originally had a program *leave*; you would run it with a time, and it would start prompting you at your terminal a little before the event, becoming more and more obnoxious as it got later. You can make `alert` serve a similar func-

tion by using the `at` facility. Instead, make a graphic `leave`, incorporating the `at` functionality.

3. Write a program like `dbdescribe` which prints out a definition for a single field or table from a schema. Try for a command line like this:

```
dbwhat [schema] {field|table}
```

4. If you have a cell phone or pager with email access, modify `dbemail` to send a message to it.
5. Make `dbemail` create a more complete report to send, including perhaps a record section plot from `dbpick`, or the actual waveforms using `trexcerpt` with the `-A` option.

Programming in C is more complex than Tcl/Tk or Perl. Programs must be compiled, often from multiple text files, then linked together with system libraries. This complexity has spawned a variety of special tools; besides the compiler and linker, one must become familiar with `make` and Makefiles, with debuggers, with syntax checkers like `lint`, and program beautifiers like `indent`. Antelope adds to this complexity, primarily with a number of new libraries, but also with special Makefile constructions, and a few programming tools.

Using Antelope in your programming is mostly about learning the libraries available, but this chapter concentrates more on the craft of C programming. A later chapter introduces the Antelope libraries.

Templates

The way to learn programming is to write programs. Sometimes it's hard to take the first step; that's where the directory `$ANTELOPE/data/templates` may be useful. This directory contains a variety of simple templates from which to start your program: `simple.c`, `dbsimple.c`, `orbsimple.c`, and `trsimple.c`. Even experienced programmers still find these useful as starting points.

You can just copy a template from the directory, but the script `template` is a bit easier, e.g.:

```
% template simple.c > example.c
```

You can add your own templates to the directory; forward them to BRTT if you'd like to see them added to the release.

Minimal Makes

Now you have the beginnings of a very simple program, but for any program you write using Antelope libraries, you will want a Makefile, and it's a good idea for every program you write, even Perl scripts. So, get a template Makefile:

```
% mkmk - > Makefile
```

This copies Makefile from the templates directory, but eliminates a lot of comments. The resultant Makefile still has way too many lines -- most Antelope Makefiles have perhaps 10 lines -- but it shows all the important options which you can set.

If you enter the command `make example` now, you'll discover that there are some unresolved references in `example.c`. Fix that by editing `Makefile`, adding `$(STOCKLIBS)` on the `ldlibs=` line. Now type `make example` and your program will be created. Add `example` to the end of the `BIN=` line, and you can just type `make`. Here's what your minimal Makefile might look like, after you've removed all the other useless lines:

```
BIN=example
ldlibs=$(STOCKLIBS)
include $(ANTELOPEMAKE)
```

This tiny Makefile is well worth the effort. Even large programs generally don't need much more complexity.

A man page would be a good idea, and there's a template for that also. The script `manpage(1)` fetches that template. When you're ready to install the man page, add one more line: `MAN1=example.1`.

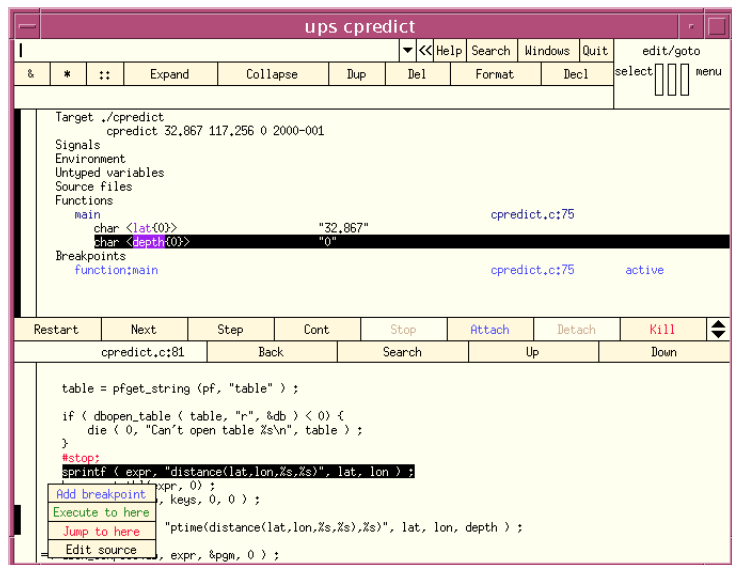
Debugging

So far this example program doesn't actually do anything useful; you have to add that code. Probably soon after you start adding code, you will be trying to figure out why `example` doesn't work as you intended. And at this point, you'll probably fire

up some debugger, either Sun's workshop or maybe gdb. Here's an alternative: the best debugger around is available free from:

<http://www.concerto.demon.co.uk/UPS/>

ups is available for at least SPARC Solaris and Intel Linux, but not Intel Solaris. It is solely a graphical debugger; there is no underlying command line interface, and the graphical interface is properly designed: small, quick, and economical. It does not have the thread debugging features of workshop, nor the memory error checking which Purify provides, but for general purpose debugging, it's unbeatable.



C allows dynamically allocating memory, rather than having arrays of a fixed size, you allocate the memory as you need it. This flexibility is absolutely essential for most complex problems. However, it's also a very common source of bugs.

Sometimes the program attempts to write outside the bounds of its allocated memory; in other cases, it never gives back (frees) any memory it allocates. In the latter case, the process just keeps using more memory until it can get no more. These kinds of problems are notoriously difficult to fix. Oftentimes, the program fails mysteriously, far from the original cause, maybe in a place you thought it could never get to. However, some useful tools exist.

- Try setting the environment variable `LD_PRELOAD` to `watchmalloc.so.1` on Solaris, or `libefence.so.0.0` on Linux, before running your program. This may expose some problems you wouldn't otherwise notice, at the actual problem.
- Sun's workshop has two checking tools under the Checks menu of the debugger: `Memuse` checking, and `Access` checking. When these work, they are very handy.
- The commercial program `Purify` from Rational Software requires that you compile your program with a special command. The resulting executable catches many different kinds of problems. You can run the executable under `ups` or `gdb` or `dbx`, put a break at `purify_stop_here`, and catch your program in the act. This is very handy, but not cheap.

Each of these programs tends to find a slightly different set of problems, though there's a large overlap.

Fixing problems before execution

It's possible to avoid a lot of debugging by catching errors with `lint(1)`. For instance, `lint` alerts you to unused variables, routines without prototypes, some variables which are uninitialized, some questionable constructions, unused arguments in subroutines, subroutines which don't set their return values, and others. Sometimes these problems are harmless, but often they indicate a problem which is a lot easier to find and fix before compilation than during execution.

```
% lint *.c
variable unused in function
    (53) n in pktchanbuf_add
(250) warning: out of scope extern and prior uses redeclared as
static: pktchanpipe_pop_pvt
argument unused in function
    (354) pvt in comp_chan
implicitly declared to return int
    (231) pktchanpipe_pop_pvt
(84) warning: variable may be used before set: mtfifo
(232) warning: constant in conditional context
argument unused in function
    (311) sig in signal_handler
function falls off bottom without returning value
```

(481) `mtfifo_release_block`

All professional programmers carefully arrange their source code to convey information about organization and flow of control, and to avoid visual clutter. `indent(1)` is a program to arrange code neatly. It has multiple options for arranging curly brackets, spaces and commas, and is very useful for cleaning up old code.

In the same vein, the program `protoize(1)` which comes with the GNU compilers is very useful for converting old common C code with separate argument declarations to ANSI C code with integral argument declarations. There's very little reason to have common C routines any more, as almost all compilers support ANSI C. Using prototypes and ANSI C declarations eliminates a whole class of errors.

If you're writing a library, you may find the program `cextract` helpful for extracting a set of prototypes from a source file. These should then go into the include file for your library.

grepsrc

When you have a large collection of source code, it's sometimes hard to remember where things are, or are used. `grepsrc` is a general purpose program which looks through certain files in a directory hierarchy trying to find a specific name. If you search for `x`, it doesn't find `xx` nor `ax`, and it doesn't look in `.o` or `.a` files (by default, at least).

Suppose you want to find an example of using a particular call, or need to change all instances of some construction, or want to find where something is defined in `/usr/include`. `grepsrc` can help in all such searches.

tag files

The program `ctags` can be used to construct an index on C and C include files. If you create a central tag file and let `vi` know where it is, then you can from within `vi` immediately go to the file which implements a subroutine from the file you're editing. Position the cursor on the first letter of a subroutine name, then press `Control-]`. Press `Control-t` to return. This is often handy.

Examples

Like all the examples in this document, the actual source code for these examples may be found under the examples directory of your Antelope distribution. There you will also find Makefiles which are slightly less trivial than the Makefiles for Tcl/Tk and Perl programs. This is largely because they specify the libraries to which these programs must be linked.

cpredict

This example takes the coordinates of an event from the command line and prints the predicted p-arrival time for a set of stations from a site table, in order of arrival. The complete source can be found in examples/predict, but the guts of the C version are shown below.

The C version of this program introduces a variety of special structures. A database pointer is kept in a Dbptr struct in C. The string expression on which the site table is sorted must be created with `sprintf`, and then put into a special Tbl list `keys`. The expression which is evaluated for each record of the site table is compiled into the Expression structure `pgm` before the loop, and then evaluated during the loop.

```
Dptr db ;
Tbl *keys ;
Expression *pgm ;
if ( dbopen_table ( table, "r", &db ) < 0 ) {
    die ( 0, "Can't open table %s\n", table ) ;
}
sprintf ( expr, "distance(lat,lon,%s,%s)", lat, lon ) ;
keys = strtbl(expr, 0) ;
db = dbsort( db, keys, 0, 0 ) ;
sprintf ( expr, "ptime(distance(lat,lon,%s,%s),%s)", lat, lon,
depth ) ;
dbex_compile(db, expr, &pgm, 0 ) ;
dbquery(db, dbRECORD_COUNT, &n ) ;
for ( db.record=0 ; db.record<n ; db.record++ ) {
    dbex_eval (db, pgm, 0, &pred ) ;
    pred += epoch ;
    dbgetv(db, 0, "staname", staname, 0 ) ;
    printf ("%5s %s (%s)\n", staname,
        s1=strtime(pred),
```

```
        s2=zepoch2str(pred, "%H:%M:%S.%s %z", "");  
        free(s1) ; free(s2) ;  
    }  
    dbex_free(pgm) ;
```

In C, one must keep track of allocated memory and free it when done, leading to various calls to free.

dbmatches

dbmatches is a companion routine to *dbjoin*. Sometimes, it is more convenient or useful to find the set of records which match some input parameters, rather than form a join. This is especially true when the parameters or the table are dynamically changing. Since a view is static, dynamic tables or parameters require repeated joins.

The idea of *dbmatches* is straightforward: it returns a list of all the records in a second table which match -- according to some join criteria which you specify -- a single record from a (probably different) table. This single record may be an actual record in a table, or it may only be a scratch record from the table, which you have filled out with the keys you are using.

This program is a simple example of how to use *dbmatches*. It prompts for a station and time range, and then prints out the records from the *wfdisc* which match those specifications. The complete program is found in `examples/c/datascope`; highlights appear below:

There are multiple structure declarations: database pointers (`Dbptr`) and lists (`Tbl *`). The new one here is the *hook*. It is used to preserve state information for the *dbmatches* routine between calls; it caches information about the join keys and index so that *dbmatches* calls are faster. Each *dbmatches* call with different tables or join keys requires its own hook.

```
Dbptr  db, dbk ;  
char   *database ;  
double time, endtime ;  
Tbl    *keys, *list ;  
Hook   *hook = 0 ;
```

Antelope programs should always call `elog_init` at the beginning, to set up some routines to trap bus errors and similar problems. Under Solaris at least, this will cause a stack trace if something disastrous occurs.

```
elog_init ( argc, argv ) ;
```

`dbopen_database` accepts arguments like `demo.wfdisc` (a table) or `-` (to read a view from `stdin`).

```
nrecords = dbopen_database ( database, "r", &db ) ;
```

In this case, we build the record which we match against in the scratch record.

```
dbk = db ;  
dbk.record = dbSCRATCH ;
```

Here, we're making up the join keys arguments for `dbmatches`.

```
keys = strtbl("sta", "time::endtime", 0 ) ;
```

Fill in the keys in the scratch record. And get back a *list* of matching records from the table.

```
for ( ;i ) {  
    dbputv (dbk, 0, "sta", sta, "time", time, "endtime", endtime,  
    0 ) ;  
    db.record = dbALL ;  
    nrecords = dbmatches ( dbk, db, &keys, &keys, &hook, &list ) ;  
    printf ( "%3d matches:\n", maxtbl(list) ) ;
```

`dbmatches` uses the pointers which make up a Tbl list as simple integers: the matching record numbers.

```
for ( i=0 ; i<maxtbl(list) ; i++ ) {  
    char rsta[32], *s1, *s2 ;  
    double rtime, rendtime ;  
    db.record = (int) gettbl(list, i) ;  
    dbgetv ( db, 0, "sta", rsta, "time", &rtime,  
    "endtime", &rendtime, 0 ) ;
```

And finally print the results.

```
printf ( "#%3d %-8s %s %s\n",  
        db.record, rsta, s1=strtime(rtime),  
        s2=strtdelta(rendtime-rtime)) ;
```

```
        free(s1) ; free(s2) ;
    }
}
```

dbevent_alarm

This program is very similar to the dbemail Perl example; it watches an origin table and sends email whenever a record is added. The core of the program is the following loop:

```
dbquery (db, dbRECORD_COUNT, &n);
for (;;) {
    sleep (sleeptime);
    dbquery (db, dbRECORD_COUNT, &n2);
    if (n2 > n) {
        sendmail (db, n, n2, email, flags);
        n = n2;
    }
}
```

It simply checks the number of records with dbquery, and calls sendmail if there are more rows.

The sendmail subroutine sends email with any new records. It opens a temporary file, writes the body of the mail there, and then runs mailx(1) on that file to send mail.

```
void
sendmail (Dbptr db, int n, int n2, char *email, Flags flags)
{
    filename = tmpnam (0);
    file = fopen (filename, "w") ;
    for (db.record = n; db.record < n2; db.record++) {
        if (dbgetv (db, 0, "lat", &lat, "lon", &lon, "depth",
                    &depth, 0) == dbINVALID) {
            complain (0,
                "couldn't read record #%d from origin table",
                db.record );
        } else {
            fprintf (file, "%5.1f  %8.3f %8.3f  %6.2f\n",
                mb, lat, lon, depth);
        }
    }
}
```

```
    }  
  }  
  fclose (file);  
  sprintf (cmd, "mailx -s 'new origin' %s < %s",  
          email, filename);  
  result = system (cmd);  
  unlink (filename);  
}
```

trload

trload is a brief example of using the trace library to read some waveform data from a database, eliminate any data flagged with missing data values, splice together any data segments split for some reason in the database, display the traces in a dbpick like window, and then save them all to a new database.

Prototypes and structure definitions for the trace library are in tr.h.

```
#include "tr.h"
```

The cbanner(3) routine is to encourage some standard information in a usage line, particularly for contributed programs.

```
static void  
usage ()  
{  
    cbanner ( "$Date: $",  
            "[ -d] [ -o dbout] [ -s subset] [ -v] db time endtime",  
            "Author",  
            "Company/University",  
            "email@domain" );  
    exit (1);  
}
```

trload uses a parameter file which specifies a Datascope view to load. In C, one begins by reading the parameter files into a Parameter file Pf * structure. Then the desired elements are extracted with a variety of pfget calls. Below, a list which describes the desired input view is taken from the parameter file.

```
Pf          *pf;  
if (pfread (Program_Name, &pf) != 0)  
    die (0, "Can't read parameter file\n");
```

```
input = pfget_tbl ( pf, "view" ) ;
```

Next, `dbprocess` takes the list and constructs the view. Then if there was a command line subset argument, `dbsubset` is then run to select out the rows of interest.

```
db = dbprocess ( db, input, 0 ) ;
if ( subset )
    db = dbsubset ( db, subset, 0 ) ;
```

The actual waveform data is loaded with `trload_css`. The data and associated parameters are read into a Datascope transient (virtual) *trace* table in memory. The table contains references to the waveform data, which is also read into memory. Notice the required initialization of the database pointer `tr` before the call to `trload_css`.

```
tr = dbinvalid() ;
if ( trload_css ( db, start, stop, &tr, 0, 0 ) )
    die ( 1, "trload_css failed" ) ;
```

`orb2db(1)` is commonly used to create the waveform files and database for a real time system. It may fill small gaps in the waveforms with special *missing* data values, to avoid creating large numbers of `wfdisc` records. `trsplit` eliminates segments in the waveforms, in the process creating more data segments and more records in the trace table.

```
if ( trsplit(tr, 0, 0) )
    complain ( 0, "trsplit failed" ) ;
```

`trsplice` performs a complementary service; adjacent waveform segments may have been split during the process of writing the database, either on preset (eg, day) boundaries, or because the system was shutdown and restarted. `trsplice` finds adjacent segments which fit together and splices them, resulting in fewer records in the trace table.

```
if ( trsplice(tr, 0, 0, 0) )
    complain ( 0, "trsplice failed" ) ;
```

The next segment illustrates the use of `dbselect` to print out a series of values for each record in the trace table, and also how to obtain the references to the actual waveform data from the trace table.

```
fields = pfget_tbl ( pf, "fields" ) ;
dbselect ( tr, fields, stdout ) ;
dbquery ( tr, dbRECORD_COUNT, &nrecords ) ;
```

```
for (tr.record = 0 ; tr.record < nrecords ; tr.record++) {
    int i ;
    float *data ;
    dbgetv(tr, 0, "data", &data, 0 ) ;
    for ( i=0 ; i<5 ; i++ ) {
        printf ( " %12.3f", data[i] ) ;
    }
    printf ( "\n" ) ;
}
```

The waveforms in a trace table can be immediately displayed in a `dbpick` type window with `trdisp`:

```
trdisp (tr, "example display" ) ;
```

Saving the data is equally simple; note that you can choose the datatype and the waveform file naming convention with `trsave_wf`.

```
if ( output ) {
    Dbptr dbout ;
    if ( dbopen(output, "r+", &dbout) )
        die ( 1, "Can't open output database %s", output ) ;
    dbout = dblookup ( dbout, 0, "wfdisc", 0, 0 ) ;
    if ( trsave_wf ( tr, dbout, "sd", 0, 0 ) )
        die ( 0, "trsave_wf failed" ) ;
}
```

It's important to set `tr.table` to `dbALL` before calling `trfree`, otherwise only one table is freed. Parameter files should be freed also.

```
tr.table = dbALL ;
trfree(tr) ;
pffree(pf) ;
```

trrotate

Gary Pavlis wrote a routine to rotate 3 component waveforms. This shows how to use it, and is very similar to `trload` above. First, waveform data is loaded with `trload_css`. Then, `trapply_calib` multiplies all the traces by `calib`, so that they're in physical units rather than counts.

```
if ( trapply_calib ( tr ) )
```

```
die ( 0, "trapply_calib fails" ) ;
```

There are two input parameters, the rotation angles `phi` and `theta`, and the routine `trrotate` handles the rest of the details. Some heuristic rules are applied to decide what traces are from the same sensor. If the routine doesn't recognize three components, those traces are not returned in the result.

```
result = trrotate ( tr, phi, theta, newchan ) ;
if ( result < 0 ) {
    die ( 0, "rotate_to_standard fails" ) ;
} else if ( result > 0 ) {
    complain ( 0, "some channels not rotated." ) ;
}
```

reading and writing orb packets

Programs which use the orb have two general problems to address:

- reading and/or writing packets
- decoding or encoding packets

These problems are handled by fairly simple interfaces in two different libraries. `orbreadpx.c` and `orbputx.c` illustrate orb readers and writers.

To begin, `orbreadpx` connects to the orb using `orbopen`.

```
if ( (orb = orbopen ( orbname, "r&" )) < 0 )
    die ( 0, "Can't open ring buffer '%s'\n", orbname ) ;
```

It may wish to preselect the packets of interest, using a couple of regular expressions which the `orbserver` applies to packet source names. Only packets whose source name matches the match expression and does not match the reject expression are sent back to `orbreadpx`.

```
if ( match ) {
    if ( (nmatch = orbselect ( orb, match )) < 0 )
        complain ( 1, "orbselect '%s' failed\n", match ) ;
    else
        printf ( "%d sources selected after select\n", nmatch ) ;
}
if ( reject ) {
    if ( (nmatch = orbreject ( orb, reject )) < 0 )
        complain ( 1, "orbreject '%s' failed\n", match ) ;
}
```

```
        else
            printf ( "%d sources selected after reject\n", nmatch) ;
    }
```

orbread may then want to reposition the read pointer to the orb, to some location other than the leading edge (where new packets are deposited).

```
    if ( after > 0 ) {
        if ((pktid = orbafter ( orb, after )) < 0) {
            complain ( 1, "orbafter to %s failed\n", s=strtime(after)
        ) ;
            free(s) ;
            pktid = orbtell ( orb ) ;
            printf ("pktid is still %#d\n", pktid ) ;
        } else
            printf ("new starting pktid is %#d\n", pktid ) ;
    }
```

orbseek positions the read pointer to a specific packet number, but that's used primarily in conjunction with state files. An orb program may be shut down at any time for some sort of maintenance. When it comes back up, it may need to start at the same packet, to avoid losing data. A state file saves the position information, so that the program can attempt to restart at the same location. When orbread is run with a -S option to specify a state file, it calls exhume during initialization to check for an old state file -- and also to setup a new state file. And then orbresurrect attempts to reposition the read pointer to the last packet successfully processed, as represented in the state file.

```
    if ( statefile != 0 ) {
        char *s ;
        if ( exhume ( statefile, &abort, RT_MAX_DIE_SECS, 0 ) != 0 ) {
            elog_notify ( 0, "read old state file\n" ) ;
        }
        if ( orbresurrect ( orb, &pktid, &time ) == 0 ) {
            elog_notify ( 0, "repositioned to pktid %#d @ %s\n",
                pktid, s=strtime(time) ) ;
            free(s) ;
        } else {
            complain ( 0, "resurrection unsuccessful\n" ) ;
        }
    }
```

The main loop of the processing is a call to `orbread`, with some processing when a packet is returned.

```
while ( totpkts < n2read && ! abort ) {
    r = orbread ( orb, &pktid, srcname,
                &time, &packet, &nbytes, &bufsize ) ;
    if ( r < 0 ) {
        complain ( 1, "\norbread fails\n" ) ;
        break ;
    }
    if ( verbose )
        processPkt ( pktid, srcname, time,
                    packet, nbytes, mode ) ;
    totbytes += nbytes ;
    totpkts++ ;
}
```

If we're using a state file, we need to save the data before exiting:

```
if ( statefile )
    bury() ;
```

Sometimes, `bury` is called within the main loop as well; `exhume` sets up some interrupt handlers, so that even when `rtexec` kills the program, it should save its current state before dying, unless it's killed with a `-9` flag.

The processing which is done for every packet needs to decode the packet first. Regardless of the packet, this is done with a call to `unstuffPkt`:

```
type = unstuffPkt ( srcname, time, packet, nbytes, &unstuffed ) ;
```

The caller should know from the returned `type` whether the packet contents are of interest. Packets that are not of interest should be quietly ignored. Packets which can't be unstuffed should probably provoke a complaint, and perhaps a complete dump of the packet.

```
switch (type) {
    case Pkt_wf :
        printf ( "waveform packet: %s\n", srcname ) ;
        break ;
    .
    .
}
```

The packet data is unstuffed into a `Packet` structure, and the routine takes parameters directly from the `this` structure. The elements of the structure which are filled in depend on the packet contents.

- A waveform packet contains waveforms and associated parameters like `nsamp` and `samprate` for one or more channels of data. The unpacked `Packet` structure has a list of `PktChannel` structures which holds the information in more convenient format.
- A database packet contains a complete database row, and potentially a complete external file referenced by that row. The corresponding unpacked `Packet` structure contains a database pointer which points to a scratch record in a virtual database.
- A parameter file packet contains a parameter file in plain string format. The corresponding unpacked `Packet` structure contains a pointer to a `Pf *` parameter file struct.

The procedure for writing to an orb is very similar. The first step is to open the orb, but with write permissions:

```
if ((orb = orbopen (out, "w&")) < 0)
    die (0, "Can't open output '%s'\n", out);
```

Now we must create a `Packet` structure, fill it out properly, and call `stuffPkt` to encode the data into an orb packet. First the static portions outside the loop:

```
pkt = newPkt ();
strcpy (pkt->parts.src_net, "BRTT");
pkt->pkttype = suffix2pkttype ("ch");
```

Inside the loop, we create a simple string packet with the current time:

```
for (;;) {
    t = now ();
    if (pkt->string)
        free (pkt->string);
    pkt->string = strtime (t);
    pkt->string_size = strlen (pkt->string);
    if (stuffPkt (pkt, srcname, &time, &packet,
                &nbytes, &packetsz) < 0)
        complain (0, "stuffPkt routine failed for %s",
                pkt->pkttype->name);
}
```

Then put the packet on the orb, wait awhile and do it again:

```
if (orbput (orb, srcname, time, packet, nbytes) < 0) {
    complain (0, "Couldn't send packet to %s\n", out);
    break;
}
tnext = floor (t + dt);
wait_for_time (tnext, verbose > 1);
}
```

Summary

Programming in c leads to greatest flexibility and the greatest access to Antelope libraries. However, it generally requires more effort and has more pitfalls. C is usually the choice when other languages have been ruled out, either for speed or because some interfaces are not available.

Exercises

1. Write a program which reads key-value pairs from stdin, creates a map using an Arr * object, and a reverse map, and then prints out both maps in alphabetical order. How does your implementation handle duplicates or replacements?
2. Write a program to examine database tables with dir/dfile external references, calculate the absolute path and the relative path (from the table) and put back the shorter path.
3. dbcp sometimes leaves unnecessarily long paths to the waveforms when it copies the waveforms. For instance, try copying the demo database to your home directory with:

```
% dbcp -f /opt/antelope/data/db/demo/demo .
```

The waveforms end up in a subdirectory *opt/antelope/data/db/demo/...* Write a program to repair this problem using *trwfname*, moving the waveform files up to the new standard directory paths 1992/138/.

4. Write some of the examples above in Perl and Tcl. What's easiest?

Makefiles are one of many great innovations from the Unix world. They provide a way of automating and documenting the construction of programs and systems, solving an otherwise knotty problem. The entire Antelope distribution is constructed with a collection of Makefiles. If you work in C or fortran, you will quickly find them indispensable.

The underlying idea and the grammar of a Makefile is quite simple. There are two primary kinds of statements:

- a macro associates a string with a name
- a rule specifies a target, some things on which it's dependent, and some instructions to execute to create the target.

```
OBJS: a.o b.o c.o
target : $(OBJS)
cc -o target $(OBJS)
```

The macro statements are straightforward, and serve much the same purpose as `#define` does in C. One complication is that when expanding a parameter, you must surround the parameter names with parens (as in the example above).

Rules are a bit more complex. The make program inspects the modification times on each of the dependencies; if the target is newer than the dependencies, make is done. Otherwise, make executes the commands on the following lines; these com-

mands are recognized because the first character is a tab. You can find better, more detailed descriptions of Makefile syntax in many books, and in the man page for `make(1)`.

Make comes with a set of built-in rules, to which are added the contents of your Makefile. This often means that you can use `make` without even constructing a Makefile. If your program is contained in a single file, say `example.c`, and doesn't require any libraries beyond the standard `libc`, you can type `make example`, and the program will be created.

Antelope provides some additional rules which considerably simplify generating programs for the Antelope hierarchy. Unlike `make`'s standard rules, these must be explicitly included, with an `include $(ANTELOPEMAKE)` statement in the Makefile. `$ANTELOPEMAKE` contains primarily rules for installing programs, man pages, and associated files into the standard Antelope hierarchy; it also provides macros for certain common sets of link libraries which simplify your Makefile.

A simple Antelope Makefile defines the program name, the man page name, the libraries which are to be linked into the program, and finally the include statement.

```
BIN=example
MAN1=example.1
ldlibs=$(TRLIBS)
include $(ANTELOPEMAKE)
```

This Makefile knows the following targets:

- `make`
creates the program locally
- `make install`
creates and installs `example` into `$ANTELOPE/bin`,
`example.1` into `$ANTELOPE/man/man1`
- `make clean`
removes any object files or created executable.
- `make uninstall`
removes `example` from `$ANTELOPE/bin`
and `example.1` from `$ANTELOPE/man/man1`

This Makefile may do a bit too much, as it descends into subdirectories and tries to run `make` there. This can be forestalled by adding a line `DIRS=` following the `include $(ANTELOPEMAKE)` line. In situations where this is desired, you may also choose the order in which subdirectories are entered by specifying a list with the `DIRS` macro.

In cases where the program is made from more than one source file, the Makefile must include a rule for making the program. There's a standard syntax for such rules which is good to follow, but you may want to use an example to avoid typos. A template Makefile with many comments is found in `$(ANTELOPE)/data/templates/Makefile`. You can get this with fewer keystrokes using the program `mkmk(1)`, and if you add an argument onto the `mkmk` command line, it strips out all the comments:

```
% mkmk - > Makefile
```

This leaves you with a Makefile where you must fill in the blanks, and delete the extra lines. For a C program, you should find a rule like this:

```
OBJS=
$(BIN) : $(OBJS)
        $(RM) $@
        $(CC) $(CFLAGS) -o $@ $(OBJS) $(LDFLAGS) $(LDLIBS)
```

Fill in the list of object files following the `OBJS=` to make the rule for your program, and delete the other rules for making a fortran program or a library.

When a directory contains more than one program, you can just put multiple entries onto the `BIN=` line; the same thing applies to most other macros.

In addition to the macros for `BIN` and `MAN1`, there are macros for `MAN3`, `MAN3F`, `MAN3P`, `MAN3T`, `MAN5`, and `MAN8`, which install the corresponding man pages. Similarly, there are macros for `LIB` and `DLIB`, should you create static libraries or dynamic libraries, and `INCLUDE` for the associated include files. The macro `PF` defines parameter files which should be installed into `$(ANTELOPE)/data/pf`. Files which should be installed into `$(ANTELOPE)/data` -- like the map database or travel time data -- can be specified using `DATADIR` to specify the directory under `$(ANTELOPE)/data`, and `DATA` to list the individual files to place there.

Install and related problems

Different versions of Unix have several different related but incompatible versions of a program `install` which is intended for copying files from a source directory

to an installation directory. This is a pretty trivial task, but it's useful to have a program which is careful about permissions and understands how to make directories. The program `deposit` serves that purpose for Antelope. You should not need to bother about it, but it is used in `$(ANTELOPEMAKE)` for the install rules.

There's a related problem which has to do with scripts: shell scripts, Tcl/Tk scripts, Perl scripts, and others. The first few lines of these scripts are typically boilerplate, but occasionally need to be changed. Rather than change them individually in every script, the boilerplate is kept in the directory `$(ANTELOPE)/data/templates`, and the script `produce` combines the boilerplate with the body of the script to create the actual executable.

(`produce` may also run `getid(1)` on the body of the script. `getid` replaces certain phrases it recognizes -- like `$(Version)` -- with new text related to the particular release. This is probably of interest only to BRTT).

Linking your program

You use Antelope libraries by linking your program with them; you can specify the libraries to link by defining the `ldlibs` macro. There are a lot of libraries available in Antelope, and many are dependent on other libraries. And of course, there are multiple system libraries. The result can be a certain confusion about what libraries you should link with your program.

For instance, for a simple program using just the Datascope interface, one possible `ldlibs` line would be:

```
ldlibs=-lds -ltrvltm -lresponse -lcoords -lstock -lbanner  
-lposix4 -lnsl -lsocket -lm
```

This shows how complex it can be to link even a simple program. Another problem is that this may change in the next release, and is certainly different in Linux. Fortunately, there's a simple alternative in this case:

```
ldlibs=$(DBLIBS)
```

This macro and a number of others are defined in `$(ANTELOPEMAKE)`; their definition changes as necessary from one release to the next, or from Solaris to Linux. There are a number of other macros as well; they are all described in the man page `antelopemakefile(5)`.

But there's still the question of what macro to use. An easy way to discover the libraries needed is to use the `ldlibs(1)` program. Given a set of binaries, it sug-

gests a `ldlibs` definition which should work. Here's an example, from the `orbputx.c` in `examples/c/orb`.

```
% ldlibs orbputx.o
$(ORBLIBS)
```

Occasionally `ldlibs` doesn't work. When this happens, it's usually one of two problems:

- there's an undefined subroutine
- a bunch of extra libraries get pulled in because a subroutine is defined in multiple libraries.

`ldlibs` has a number of other useful options. For instance, it can tell you where a subroutine is defined:

```
% ldlibs -w htonl
htonl is found in socket xnet
```

Read the man page for more information.

Points of Confusion

The lines in a rule which may be executed must begin with a tab. Since a tab normally looks just like several spaces, sometimes it's typed in that way, and then `make` fails.

`make` reads (by default) one file named either `makefile` or `Makefile`. If both are present, then the `makefile` is read. However, it's common to use `Makefile` exclusively, to highlight the file in an `ls` listing. `makefile` tends to get lost in a sea of files.

Conclusion

This is a very brief introduction to the `make` program and `Makefiles`, but it should be adequate for most simple programs. Few of the programs in *Antelope* require any more complexity in their `Makefiles`. If you need more, there is (of course) a man page on the rules: `man antelopemakefile`. You can also look at the text file itself: `vi $ANTELOPEMAKE`. There is also a man page for `make(1)`, and many books have a section on `make`.

Exercises

1. Write a Makefile to install your C program into the `$ANTELOPE` hierarchy.
2. Add your parameter file and man page to the Makefile above.
3. Write a Makefile for one of your scripts. Remove the “boilerplate” portion and rename the body `x.tcl`, `x.wish`, `x.pl`, or `x.sh` for a Tcl, wish, Perl, or shell script, respectively.

The bulk of Antelope software development toolkit is C libraries. This chapter provides an overview and brief introduction to these C libraries. Detailed documentation on the routines is found in the man pages. The *Antelope Programmer Reference Guide* also contains brief synopses for these libraries.

Throughout this chapter, the libraries are referred to by their root name, as you would refer to them on a link line: *-lroot*. The corresponding full library name is either `$ANTELOPE/lib/libroot.so` or `$ANTELOPE/lib/libroot.a`.

Stock Library

The stock library contains a smorgasbord of small, but useful routines.

Lists

Lists are native to Perl and Tcl/Tk. A similar facility is implemented for C with the *Tbl* object. A new list is created by `newtbl`, objects are added with `settbl` (or `pushtbl`), objects are deleted with `deltbl` (or `poptbl`), and the whole works can be freed up with `freetbl`.

```
Tbl *t ;
t = newtbl(5) ;
for (i=0 ; i<10 ; i++ ) {
    sprintf ( s, %d, i ) ;
    settbl ( t, i, strdup(s) ) ;
}
s = deltbl ( t, 8 ) ;
free(s) ;
freetbl(s, free) ;
```

One need not worry about size, as the list grows as necessary to accommodate its contents. (However, the space used never shrinks). One can use lists for stacks, FIFOs, LIFOs, and similar structures.

The ring routines (`newring`, `pushring`, `getring`, ...) implement simple ring buffers on disk using memory mapped *Tbl* lists. Similarly, bit vectors are a natural extension using *Tbl* lists. As you might expect, one can create a bit vector, set and test various bits, and perform some logical operations (and, or, xor) between multiple bit vectors.

```
Bitvector *b ;
b = bitnew() ;
for ( i=0 ; i<30 ; i+=2 ) {
    bitset(b, i) ;
}
```

Associative Arrays

In associative arrays, the indices are character strings, rather than integers. This concept is implemented as part of the language in `awk`, `Tcl/Tk` and `Perl`. It is very useful in C as well, but more clumsy. New arrays are created by calling `newarr`, new elements are added by `setarr`, deleted by `delarr`, and the whole caboodle freed with `freearr`:

```
Arr *aa ;
char *s ;
a = newarr(0) ;
setarr(aa, hi, strdup(there)) ;
s = getarr(aa, hi) ;
s = delarr(aa, none) ;
```

Associative arrays are represented as balanced binary trees, with each node containing a string key and a pointer to the associated value.

Stbl

Sometimes you may have objects (structures) which you would like to organize by a binary key, without making a string key. In this case, the *stbl* routines are of interest. You must provide a comparison routine which takes pointers to two of your structures and indicates the order in which they should be put. (It is essential that the concept of order make sense for your objects). Then the *stbl* routines can be used to organize the structures into a balanced binary tree. Below, a new *stbl* is created with *newstbl*, specifying *cmp* as a comparison routine. An entry is added with *addstbl*, which may return a pointer to a deleted entry. One can then walk through the entries in order with *getstbl*(5)

```
stbl = newstbl (cmp) ;
if ( (ep = (Entry *) addstbl ( stbl, e )) != e )
    free ( e ) ;
n = maxstbl ( stbl ) ;
for ( i = 0 ; i < n ; i++ ) {
    ep = (Entry * ) getstbl ( stbl, i ) ;
```

The key is whatever parts of the structure your comparison routine uses; the value is the remainder of the structure.

Error Handling

A central facility for error handling allows some uniformity in applications, and a variety of useful features (like intercepting errors with a single breakpoint in a debugger). The *elog* routines also provide an error log, which library routines may use to leave detailed messages. The calling program decides when, if and how to display these messages. There are just a few commonly used calls to the *elog* routines: *elog_init*, *elog_complain* and *elog_die*.

elog_init is used near the beginning of the main routine to initialize the *elog* routines. It also sets up handlers for various program faults like bus error.

elog_complain notifies the user when something non-fatal has happened, whereas *elog_die* causes the program to exit.

```
elog_init ( argc, argv ) ;
```

```
file = fopen ( filename, r ) ;
if ( file == 0 ) {
    elog_die(1, Cant open %s, filename ) ;
}
```

The first argument to `elog_die` (and other similar `elog` routines like `elog_complain`) is a flag for whether to print a system error message based on the system global error flag `errno`. It's not uncommon for `errno` to be non-zero due to a much earlier failed system call; consequently, the first argument is usually zero, unless it follows a failed system call. The remainder of the arguments are like `printf(3)` arguments: a format and a list of corresponding expressions. A call to `elog_complain`, `elog_die` and related functions causes the error log to be printed also.

The behavior of the `elog` routines can be affected by certain environment variables or the parameter file `elog.pf`. Specifically, you can change where error output is directed, cause a debugger to be called when an error occurs, or prepend a time tag to error messages. These facilities are used in the real time system, and during development. See the man page for more information.

Path Manipulation

Often, it's necessary to convert a relative filename path to an absolute path, to ensure portability. `abspath` accomplishes this; `relpath` performs the inverse function, converting a local or absolute path into a relative path from a particular directory. `cleanpath` eliminates extra slashes and dot components from a path, and optionally removes any symbolic links from a path. `parsepath` separates a path into a `dirname`, `basename`, and (optionally) suffix.

Two routines, `datafile` and `datapath`, provide methods for locating program files inside the `$ANTELOPE/data` directory. This can be useful for other programs that need centrally stored files (like parameter files, travel time curves, Perl or Tk extensions, etc.). Antelope provides many other path manipulation routines, but these generally are not found in the Perl, Tcl/Tk or matlab extensions.

`makedir` creates (if necessary) all the directories along a path. `is_dir`, `is_file` and `is_nfs` return true if the path argument corresponds to a directory, a file, or is nfs mounted.

Parameter Files

The routines which implement parameter files are in the stock library. The input routines convert the string representation with which you're familiar into an internal tree structure of Pf * objects. Each node has a name, and a value. The value may be a string, a list or an associative array.

The routines `pfread`, `pfcompile`, `pfupdate`, and `pfin` are just various methods of calling the compiler which converts the external string representation of parameter files into the internal tree representation -- and sometimes, merging multiple external parameter files along the `PFPATH` into one internal parameter space. `pfread` reads all the files corresponding to a specific parameter file name, whereas `pfin` reads from an open file descriptor, and `pfcompile` converts a string already in memory. `pfupdate` checks if any parameter file corresponding to a name has changed (or sprung into existence) since the previous time `pfupdate` was run, rereading all the files when something has changed.

There are a variety of routines for traversing the internal Pf tree and retrieving values in particular formats. Things are more complex in C than in Perl or Tcl/Tk because of the strong typing in C. Ultimately, every scalar value in a parameter file object, internal or external, is a string. `pfget_double`, `pfget_int`, and `pfget_boolean` convert these strings into floating point, integer, or boolean values. The routines `pfget_tbl` and `pfget_arr` return lists and associative arrays corresponding to lists and arrays in a parameter file, provided those lists or arrays have only scalar values. It's not possible to return an array of lists, for example.

When the `pfget` routines return a string or a list of strings or an array of strings, the pointers point to strings inside the internal parameter file tree. Changing these would change the Pf object; freeing such a string would cause problems later when the entire parameter space is freed with `pf free`.

It's possible to write out a single parameter file, with the routines `pfout`, `pfwrite` or `pf2string`. In addition, one can discover the actual files which make up an internal parameter space with `pf files`.

String manipulation

An assortment of string routines augment the standard `string(3)` functions.

`strmatches` and `strcontains` check whether a string matches in its entirety or just contains a particular regular expression. This simplifies dealing with the regular expression interfaces.

The `morphtbl` routines provide a way of applying a series of regular expression pattern substitutions on an input string.

`strsub` replaces a regular expression in a string. `expandenv` looks through a string for expressions of the form `$XYZ`, replacing that with the value of the corresponding environment variable (if present),.

`strtr` translates characters in a string from one set to a different set, similar to the `tr(1)` command.

`str2upper` and `str2lower` convert strings to upper or lower case.

`strtrim` trims white space from both ends of a string.

`pushstr` and `popstr` implement a variable length string; you can add to it without worrying about the space.

Miscellany

One routine of interest is `cbanner`; it prints a standard header giving some information about a program like its version, its author, and a usage line. Contributed programs should make a point of calling this routine in response to a `-V` flag on the command line, e.g.:

```
while ((c = getopt (argc, argv, "vV")) != -1) {
    switch (c) {
        case V: {
            char *usage      = "example [-v] file" ;
            char *author     = "Your Name"
            char *location   = "Your Institution" ;
            char *email      = "you@yours.org" ;
            cbanner ( "$Revision: 1.3 $" , usage, author,
                    location, email ) ;
        }
    }
    break;;
}
```

A quicksort, a shell sort and an insertion sort are present, as well as a binary search.

Routines to test for or set IEEE 754 Standard Floating Point special numbers are not standard, so the stock library has private versions. `is_infinity` returns non-zero when its argument is an IEEE infinity; `a_nan` returns an IEEE not-a-number.

`disordered(3)` is a simple random number generator which has the virtue of giving the same results on Solaris and Linux, unlike `random(3)`.

The routines `ask`, `askyn`, and `askynaq` query `/dev/tty` (not `stdin`) for the answer to a provided prompt. `asknoecho` prompts for a password without echoing what is typed in.

`memdup` corresponds to `strdup` with a count.

Several compress/decompress routines are present: `gencompress` and `genuncompress`, used in the GENC packets, and `cm6` and `um6`, the plain ascii compression used by autoDRM. `uucsd` and `cucsd` implement a compression scheme once used by some UCSD data loggers.

`fdkey` tests whether input is available on the provided file descriptor, while `fdwait` waits some limited time for input before returning.

The `tty` routines change the standard input processing on a file descriptor among a few useful states like `echo/no-echo`, by character (as an editor like `vi` would require) or by line (the usual `@default`).

`hexdump` prints a region of memory in a standard (IBM 360) hexadecimal dump; `read_hexdump` performs the inverse operation.

`now` returns the current time, typically to high precision.

Checksums used by the Quanterra data logger and by autoDRM are implemented.

coords library

The routines of most interest in the `coords` library are related to time conversion. The routine `str2epoch` converts an input time in a variety of formats to an epoch time. Use `epoch(1)` to verify that your time expression is properly recognized.

`epoch2str` takes an epoch time and a format string (much in the style of `strftime(3)`) to generate the output time as you like. `zepoch2str` allows specifying a time zone like `US/Mountain` to get a time other than UTC. A blank time zone is understood to mean the default time zone for the computer.

Time zone names can be discovered by looking at `/usr/share/lib/zoneinfo` on Solaris, or `/usr/share/zoneinfo` on Linux. (Naturally, they're not in the same place, nor do they contain the same names).

response library

The routines of the response library can read and create the response files for CSS 3.0. These files support the commonly used stage filter types of poles and zeros, frequency/amplitude/phase (measured) filters, and Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. `read_response` reads a response file and `eval_response` evaluates the response at a particular frequency.

travel time libraries

Antelope provides a generic interface to a travel time calculator, which is documented in `ttcalc(3)`. There are two calls: `ttcalc` for travel times, and `ucalc` for slowness calculations. The input arguments are the same; the output argument is a Tbl list of either travel times or slowness. (A final argument, the hook, is actually defunct and can be ignored).

One argument, the *method*, identifies the particular calculator to use, e.g. `tau-p`. Another argument, the *model*, identifies the earth model, e.g. `IASP91`.

The `ttcalc` interface uses the method name to load a dynamic library which implements the calculation method; it passes the model name to routines in this library, which must look up the model parameters it needs.

Both calls require a 3d source and observer location. Although the existing calculators are only 2d, a 3d calculator could immediately be integrated and used.

The program `ttcalc(1)` is an interactive interface to the travel time libraries. One commonly used travel time calculator is the tau-p calculator from Brian Kennett and Ray Buland. Gary Pavlis has provided a layered plane model named `ttlz`. Other models should be easy to integrate.

trace library

The trace library is a collection of routines for reading and writing waveforms. There is an unfortunate variety of waveform formats, including many different compression schemes. In addition, there are problems with data segmentation (the waveform segment you need may span two different waveform files). Also, you may want some of the other information from the database, like `calib`, station location, or instrument response. The trace library tries to hide this complexity from the programmer.

The trace library loads waveform data into memory as an array of floats, and saves a reference to this data in a transient database table constructed for that purpose. The trace table is much like a join of the `wfdisc`, `site`, `sitechan`, `sensor`, and `instrument` tables. Parameters are read from the database table with standard Datascope calls. For a particular row of this table, one might use `dbgetv` to get `sta`, `chan`, `time`, `calib`, and `nsamp` just as with a `wfdisc` row, but also get the value of `data`, which returns a pointer to the associated array of floats.

You can inspect the Trace schema and see how the trace table is composed using the command `dbhelp Trace4.0`. You can also modify the Trace4.0 schema for your own purposes, adding fields or tables which you need for your own processing.

To create a trace table from the database, first construct a joined view which contains `wfdisc` and the other tables from which you want to load values. Eliminate stations and channels you don't need. Then call `trload_css`:

```
tr = dbinvalid() ;
if ( trload_css ( db, start, stop, &tr, 0, 0 ) ) {
    die ( 1, "trload_css failed" ) ;
}
```

This reads data from the database view `db` between time *start* and *stop*, returning a new trace table with pointers to the waveform data. You can get these pointers and other parameters from the trace table using Datascope calls:

```
dbquery ( tr, dbRECORD_COUNT, &nrecords ) ;
for ( tr.record = 0 ; tr.record < nrecords ; tr.record++ ) {
    int i ;
    float *data ;
    char sta[16], chan[16] ;
    dbgetv(tr, 0, "data", &data, "sta", sta, "chan", chan, 0 ) ;
    printf ( "%-6s %-3s\n", sta, chan ) ;
    for ( i=0 ; i<5 ; i++ ) {
        printf ( "  %12.3f", data[i] ) ;
    }
    printf ( "\n" ) ;
}
```

You would at this point apply your own processing to the data.

Once the data is represented in a trace table, it is trivial to display it with a `dbpick(1)` style window, or to save it to a new database:

```
trdisp ( tr, "example display" ) ;
if ( dbopen(output, "r+", &dbout) ) {
    die ( 1, "Can't open output database %s", output ) ;
}
dbout = dblookup ( dbout, 0, "wfdisc", 0, 0 ) ;
if ( trsave_wf ( tr, dbout, "sd", 0, 0 ) ) {
    die ( 0, "trsave_wf failed" ) ;
}
```

orb library and Pkt library

The real time system revolves around packets in the `orbserver(1)`. The `orb(3)` library provides the interfaces into the `orbserver`. The `Pkt` library provides a standard unstuff mechanism for converting packets from some generally opaque internal format to a C structure which is convenient for processing. It also provides a standard method for creating a binary packet from a C structure; it's also common for programs to create the binary packet without reference to the `Pkt` library.

Unfortunately, Antelope has both a libpkt.so and a libPkt.so. libpkt.so is the original, but now obsolete, packet library. A few programs still use it, but it is outdated. Do not get confused and do not use libpkt!

Packets on the orb are distinguished primarily by their names. Every packet has a source name, which should convey information about its source and content and the data format of the packet. The convention is that packets are named

```
net_sta_chan_loc/type/subcode
```

The *net_sta_chan_loc* portion identifies the source; the *type* identifies the packet's data format, that is, how the data is encoded within the binary packet. The *subcode* is optional, and may further qualify the packet. This source name is used by the orb interface to select and/or reject certain sets of packets, according to regular expressions which are passed to the orbserver.

In addition to the source name, each packet has a time, and the orbserver assigns it a temporary integer id, pktid, which identifies it uniquely while it's on the orbserver. After the packet has fallen off the orbserver, the pktid will be reassigned to a new packet.

There are a very limited number of orb calls. The first step is to initiate an orb connection, by calling orbopen.

```
if ( (orb = orbopen ( orbname, "r&" )) < 0 )  
    die ( 0, "Can't open ring buffer '%s'\n", orbname ) ;
```

The orbname is of the form *host:port*; however, it's possible to assign names to port numbers by adding the names to the file:

```
$ANTELOPE/data/pf/orbserver_names.pf
```

The second argument is permission, and must start with “r” for read connections, and “w” for write connections. Notice the “&” following the “r”; this ampersand engages an automatic reconnection protocol, if the connection ever fails. It's fairly common for connections to fail over a period of time, usually due to failures in the connectivity between orbserver and client machines. Of course, it's also possible for the orbserver machine to die, or for the orbserver to be shut down for some reason. In any of these cases, when automatic reconnection is requested, the client program keeps retrying indefinitely. When it succeeds, it makes every effort to resume reading from the last packet. If automatic reconnection is not requested, orb calls begin to return errors, and the client program should quit.

A program which reads from the orbserver may select only certain packets with `orbselect` and `orbreject`, providing regular expressions which match source names which should be selected or rejected.

```
if (orbselect ( orb, match ) < 0 )
    die ( 1, "orbselect '%s' failed\n", match ) ;
if ( orbreject ( orb, reject ) < 0 )
    die ( 1, "orbreject '%s' failed\n", match ) ;
```

A reading program may also reposition the read pointer, which is initially at the leading edge (the newest packet) when an orb is opened. `orb2db(1)`, for instance, normally repositions to the oldest packet.

```
if (orbafter ( orb, after ) < 0) {
    die ( 1, "orbafter to %s failed\n", s=strtime(after) ) ;
}
```

Finally, the program will work in a tight loop, reading packets with `orbread`, and processing each packet:

```
for(;;) {
    x = orbread ( orb, &pktid, srcname,
                &time, &packet, &nbytes, &bufsize ) ;

    if ( x < 0 ) {
        complain ( 1, "\norbread fails\n" ) ;
        break ;
    }
    processPkt ( pktid, srcname, time, packet, nbytes ) ;
}
```

A packet contains data that has been encoded in some format which is convenient for transmission, or for the author. Many programs must be able to extract that information, however, and the Pkt library provides a common interface for that purpose. The routine `unstuffPkt(3)` takes the information returned by `orbread(3)`, figures out which routine in the library understands how to decode the packet, and runs that routine.

Each of these unstuff routines must decode the packet data into a standard Packet structure. The Packet structure has places for all the standard information in a packet: `sta`, `chan`, `time`, `nsamp`, `calib`, `calper`, and data for each channel of a waveform packet, a `db` pointer for a database packet, a `pf` pointer for a parameter file packet, and a string pointer for a character string packet.

In addition, there are a number of user variables in which an unstuff routine may place extra information, and a hook (see `new_hook(3)`) where almost anything might be placed. Of course, the routine using that extra information must recognize when the information is present, and when it is not.

More generally, routines may need to recognize broad categories of packets. Although `orb2db` is only interested in waveform packets, it may by mischance read database packets or parameter file packets. This should not cause it to die. `unstuffPkt` returns an integer id which identifies a broad category; some of the predefined categories are *Pkt_wf*, *Pkt_db* and *Pkt_pf*. The processing loop might look like this:

```
static Packet *unstuffed=0 ;
PktChannel *achan ;
int type, *data ;
char record[STRSZ] ;
type = unstuffPkt (srcname, time, packet, nbytes, &unstuffed) ;
switch (type) {
  case Pkt_wf :
    printf ( "waveform packet: %s\n", srcname ) ;
    for ( i=0 ; i<unstuffed->nchannels ; i++ ) {
      achan = (PktChannel *) gettbl(unstuffed->channels, i);
      printf ( "%-8s %-8s %-8s %-8s\n", achan->net,
              achan->sta, achan->chan, achan->loc ) ;
      for (i=0 ; i<5 ; i++ ) {
        fprintf ( file, " %8d", achan->data[i] ) ;
      }
      fprintf ( file, "\n" ) ;
    }
    break ;

  case Pkt_db :
    dbget ( unstuffed->db, record ) ;
    printf ( "database row: %s\n %s\n", srcname, record ) ;
    break ;

  case Pkt_pf :
    printf ( "parameter file: %s\n", srcname ) ;
    pfout(stdout, unstuffed->pf) ;
}
```

```
        break ;

    default:
        printf ( "unrecognized packet type %d: %s\n",
                type, srcname ) ;
        break ;
}
```

Many orb programs only terminate when killed. Such programs may want to use the `bury/exhume` pair of routines to save and restore a state file, so that they begin reading from the same packet when they're restarted.

A program may open and close a connection, however. For instance, some programs maintain one connection for reading, and open a second connection briefly for writing.

Most programs which write to the orb operate in a tight loop with calls to `orbput`:

```
for (;;) {
    make_packet ( srcname, &time, &packet, &nbytes) ;
    if (orbput (orb, srcname, time, packet, nbytes) < 0) {
        complain (0, "Couldn't send packet to %s\n", out);
        break;
    }
}
```

Such programs need to construct the packet which is being sent to the orb. Although many programs simply hard code this, it's good to use the `Pkt` library if possible.

Using the `stuffPkt(3)` routine is fairly straightforward. First you select a packet type. For simple waveform packets, *GENC* is a good choice. For database packets and parameter file packets, the `db` and `pf` packets are the right choice, and for simple strings, the `ch` packet type. Different packets may require that you add your own routines to the `Pkt` library, as described in a later chapter.

Once the packet type is determined, the program fills in the appropriate values in a `Packet` structure, and then calls `stuffPkt` and `orbput`:

```
pkt = newPkt() ;
strcpy (pkt->parts.src_net, "BR");
strcpy (pkt->parts.src_sta, "BRTT");
```

```
strcpy(pkt->parts.src_chan,"BHZ" ) ;
pkt->pkttype = suffix2pkttype("GENC");
pkt->nchannels = 1 ;
for(;;) {
    .
    .
    if (stuffPkt(pkt, srcname, &t, &packet, &nbytes, &pack-
etsz)<0) {
        complain ( 0, "stuffPkt failed for %s\n", pkt->pkttype-
>name ) ;
    } else if ( orbput ( orb, srcname, t, packet, nbytes) ) {
        complain ( 0, "orbput fails\n" ) ;
    }
}
```

The fragment above first creates a new packet structure, then fills in a few values which are relatively static, including the packet type. It then goes into a loop of `stuffPkt` and `orbput`. In this case, it's creating a GENC waveform packet. In the empty gap, the `pkt->channels` list must be filled in with a `PktChannel` entry for the BHZ channel. Each such entry has data points, and `sta`, `chan`, `time`, `nsamp`, `calib` and `calper` parameters, and possibly others.

The `stuffPkt` routine should also be used for creating `db` and `pf` packets. The procedure in these cases is analogous, but generally simpler. It's only necessary to fill in the `db` or `pf` component of the `Packet` structure:

```
pkt->pkttype = suffix2pkttype("db");
db = dblookup ( db, database, table, 0, 0);
db.record = 0 ;
pkt->db = db ;
if(stuffPkt(pkt, srcname, &t, &packet, &nbytes, &packetsz) < 0){
    complain ( 0, "stuffPkt routine failed for %s\n",
                pkt->pkttype->name ) ;
} else if ( orbput ( orb, srcname, t, packet, nbytes) ) {
    complain ( 0, "orbput fails\n" ) ;
}
```

or

```
pkt->pkttype = suffix2pkttype("pf");
pkt->pf = pf ;
```

Other libraries

Undocumented libraries and interfaces are unsupported and may change or disappear between releases. However, there are a few libraries with limited documentation which you may find them useful.

The `pixaddress` library implements a tiling scheme for a sphere. The pixels cover the surface with approximately square and approximately equal area pixels, and have nice properties for finding nearest neighbors. There are multiple resolutions available, up to 28 bits per pixel, corresponding to about 20 arc second resolution.

The `proj` library implements a large variety of map transformations from Gerald Evenden. The user initializes a transformation by calling `proj_init` with a list of string arguments; then `proj_fwd` transforms lat/lon coordinates to map x-y coordinates, while `proj_inv` transforms map x-y coordinates to lat/lon coordinates.

The `tk` library implements a C-callable Tk send protocol; `dbpick` uses it to receive and send messages from a Tk application.

The `vogle` library is a small and relatively complete library of 3d graphics routines.

The `forb` library implements a portable file representation for orb packets. `orb2orb` can read or create such files, and most orb programs automatically read or write such files if the orb name appears to be a filename: begins with either “.” or “/”.

Summary

This quick tour of the Antelope libraries should give you a taste of what’s available. The next step is to actually begin using some of the libraries in your own programs. The man pages should prove helpful in getting the usage just right. Browsing through the Antelope Programmer Reference Guide may introduce some new interfaces not covered in this chapter, as well as remind you of the ones which are. The examples distributed with Antelope use many of these interfaces. In addition, there are programs available from the Antelope User Group web site which might be of interest.

Exercises

1. Use `ldlibs -w` to find out where the following routines are:
 `ntohl eatom abs fabs besj0_ sacfre_`

2. Use `ldlibs -r` to find out what routines used in `libtr` are resolved in `libcoords`.
3. Read the `stock.h` include file, and the macros for `allot`, `reallot`, `SIZE_BUFFER` and `RESIZE_BUFFER`.
4. Make and run the various programs in the `examples/c/datascope`, `examples/c/tr` and `examples/c/orb`.

In addition to writing your own programs using Antelope libraries, you may also extend the basic Antelope system to use your data formats or processing. For instance, you can

- extend the trace library to read and write your special waveform data format
- extend `dbloc2` to use your own location program
- extend `dbloc2` to use your own magnitude calculator
- add your own packet unstuff routines
- add your own travel time calculator
- add your own functions in parameter files (like `&exec`, `&ask`, and `&glob`)
- extend `dbe` with your own programs

Generally, you would program these extensions in C, though Tcl/Tk or Perl are also options for `dbe` extensions.

Adding new waveform types

Antelope allows reading and writing a wide variety of datatypes, but new ones appear with some regularity. If you use a different datatype and would like to read those waveforms using Antelope utilities like `dbpick` and `trexcerpt`, you can do so in a straightforward way.

You must first create wfdisc files for your waveforms; programs like sac2db, ah2db, and psd2db are standard Antelope examples. You may be able to create wfdisc files with a Perl script.

Next, you must extend the trace library to understand your data format by

- modifying a list of datatypes in the `css3.0` schema file
- modifying a list of datatypes in the parameter file `trdefaults.pf`
- creating a library `libuser.so` containing a routine which can read your data files, and return an array of floats.

The man page `addwf(5)` describes the procedure, and there is a complete example in `examples_c_addwf(5)`. The example re-implements a two byte Sun order integer format, adding a (non-functional) header and a trailer. Here's the necessary read routine:

```
int
readxx ( char *input, int nbytes, int first, int nsamp,
         float *dest )
{
    int i, retcode ;
    short x ;
    if ( nbytes >= xxHdrSz + (nsamp+first) * sizeof(short)) {
        input += xxHdrSz ;
        for ( i = 0 ; i<nsamp ; i++ ) {
            N2H2(&x, input+((i+first)*sizeof(short)), 1 ) ;
            dest[i] = x ;
        }
        retcode = 0 ;
    } else {
        retcode = -6 ; /* file too short */
    }
    return retcode ;
}
```

The routine is called with a memory buffer containing the raw data from the waveform file; the routine must read the raw data, convert it to floats and return it in the `dest` array. Actually, the entire buffer does not need to be converted, only `nsamp` samples starting at sample `#first`. The `dest` array is large enough for `nsamp` samples.

The `wfdisc` datatypes normally imply a byte order (unless the data format encodes the information internally, as `miniseed` does); the routine must handle any byte swapping necessary. In the example above, this is handled with the routine `N2H2`, which converts 2 byte Sun order integers to local host 2 byte integers. The other notable feature of this short routine is the hard-coded step over the fixed header at the beginning, and the test to ensure there's enough data provided to contain the samples requested.

If you need to write data in your format, there's a bit more work. You need to add up to three routines to `libuser`: one to write a header (like SAC or AH require), one to write the actual data, and one to write any trailing data (`autoDRM` wants a checksum). The output routine must also deal with any byte swapping required, and must also check each sample to ensure that it fits into the output format. For example, a float might not be an integer, or might be too large for a two byte short integer.

Using your location program with `dbloc2`

`dbloc2` is a collection of programs which are used to help an analyst review and locate earthquakes. The programs are bound together with a Perl script `dbloc2` and a Tcl/Tk GUI `dbloc_buttons`. By default, the user can run three different location programs: `dblocsat`, `dbgenloc`, and `dbgrassoc`. You can add your own location algorithm to this mix, by providing a version of your program which adheres to the input/output requirements of `dbloc2`.

To get a location, `dbloc2` writes a parameter file containing some standard parameters (including the set of arrivals and travel time model), and possibly some location program specific parameters. It then writes a single blank line to the stdin input of the location program. The location program understands this as a signal to begin, and it:

- reads the parameter file for arrivals and other parameters
- reads the input database for station locations
- attempts to find the event location
- writes results into the trial database
- writes a single output status line.

`dbloc2` recognizes this output status line, and signals `dbloc_buttons` to reconfigure the display appropriately.

There is a contrived example of all this in `examples/c/dbloc`. There's a program, `someLoc`, which conforms to the i/o requirements of `dbloc2`, but lacks any real location algorithm.

Once you have a program which conforms to the requirements, it's a fairly simple matter to integrate it into `dbloc2`. You must provide three simple routines in Tcl and integrate them into `dbloc_buttons` library in the directory:

```
$ANTELOPE/data/tcl/library/dbloc
```

These additional routines allow the user to interactively change special parameters for your location program, and to insert those parameters into the parameter file which `dbloc2` creates before each run. The example mentioned above contains extremely simple outlines of the Tcl routines needed.

The integration required is to copy the file(s) containing these routines to `$ANTELOPE/data/library/dbloc`, and edit the `tclIndex` file there to add these routines. Finally, you must add a line to the `location_programs` list in the `dbloc2` parameter file, specifying how to start your program and the valid travel time models for it.

Using your magnitude calculator

After every successful location, `dbloc2` calls one or more magnitude calculators, from a list `magnitude_calculators` in the `dbloc2` parameter file. These calculators are called with two arguments: the database and `orid` of an origin. The program must read the database to obtain parameters and waveforms for its calculations, and then must write the results into the database, in the `netmag`, `stamag`, and (possibly) the origin tables.

Nicholas Horn (from ZAMG in Vienna) has contributed a magnitude calculation program named `dbampmag`. This could be a very helpful starting point if you write your own; or you may find it is adequate as is. You can obtain the source code for it from the Antelope User Group web page, <http://www.indiana.edu/~aug>.

Add your own packet unstuff routines

Perhaps you have data loggers which are not already supported by Antelope or the contributed software. If you already have software which talks to the data loggers, you may be able to adapt it to write data packets to the orb and into the Antelope real time system.

There are two approaches to this:

-
- use an existing Antelope packet format (preferably GENC)
 - create a new Antelope packet format which is convenient for your data

The former approach is much simpler: your acquisition program should use the routine `stuffPkt` to repackage information from the data logger into packets, and put them onto the orb. Everything downstream will then know how to handle these packets, without any further work.

It's relatively straightforward to add your own packet, but you must extend the `libPkt` library so that it understands how to unstuff your packets, so that downstream programs can use the data. If your packets leave your site, you should provide these `libPkt` updates to downstream sites.

Regardless which approach you take, your packets should fulfill certain requirements:

- the `srcname` should follow our conventions; for data packets, that means names of the form `net_sta_chan_loc/type/subcode`. `net` is a SEED network code, `sta` is a SEED station code, `chan` is a SEED channel code, and `loc` is a SEED location code. `_loc` is omitted if there is no location code. `_chan` is omitted if the packet is multiplexed (has more than one channel). The type code is the way the unstuff routine figures out what routine to use; it must be unique. `subcode` is an optional string which may further classify the packet.
- data packets must specify station and channel codes; often the program talking to the data logger must assign these codes. In addition, data packets should contain values for `calib`, `calper` and `rsptype`. The data acquisition program usually fills these in, either from a database (preferably) or a parameter file.

If you decide to create your own packet type, then you start by copying `$ANTELOPE/data/pkt` to some directory where you can make your changes. The `pkt` directory contains the complete source for the `Pkt` library; you must add your packet type, recompile the library and install it in `$ANTELOPE/lib`. When your programs are restarted with the new library, they should understand the new packet type.

Don't modify `Pkt.h`, unless you're sure you know what you're doing. Changes in the structures or constants in `Pkt.h` could cause every program using `libPkt` to break.

All your modifications should be in the file `packets.pf`. This file contains a one line summary of each packet type, plus a prescription for creating the binary packet, and for unstuffing the binary packet into a `Packet` structure. The `packets.pf` file is

processed by `mk_libpkt(1)` to create the file `Pkt.c`, which is compiled with the other `.C` files to create `libPkt.so`. The Makefile is a fairly standard Antelope Makefile, and knows how to create and install `libPkt.so`. Your problem is to write the prescription in `packets.pf` and get it to compile and work.

There is a simple example for adding an imaginary packet containing some weather data in `examples/c/pkt`. Actually, it shows two alternatives: creating a new `libPkt.so` as described above, or adding new stuff and unstuff routines to a special `libuser.so`. The former is probably the better choice.

Add your own travel time calculator

If you already have your own travel time calculation routines, you can integrate them into Antelope so that they are used or accessible in many standard Antelope programs. The basic idea is to create a dynamic library with two entry points for calculating travel times and slowness conforming to the `trvltn` library requirements. The `trvltn` library accepts an argument called the *method* which is the root name of the library, and also the name of the travel time routine *in* the library, and part of the name of the slowness routine.

In `examples/c/trvltn`, a travel time calculator aptly named Bogus is implemented. (Don't use it for your network locations). The dynamic library is named `libBogus`, the travel time calculation routine is named `Bogus`, and the slowness routine is named `Bogus_ualc`.

The Bogus routine is easy to read; the input parameters are *model*, *phase_code*, *mode*, and *geometry*. The output is all contained in the list *times*. The *hook* is to enable the Bogus routine to cache some state data. If it needed to read a lot of tables from disk, it might save the binary image in *hook*, to avoid rereading data every call. (Bogus is much too simple to need that).

```
int
Bogus ( char *model, char *phase_code, int mode,
        TTGeometry *geometry, Tbl **times, Hook **hook )
{
    int retcode = 3 ;
```

The *model* parameter identifies a particular earth model; it could relate to a particular position on the earth.

```
    if ( strcmp(model, "1d") == 0 ) {
```

The source and receiver positions are (potentially) provided in 3d coordinates in the input TTGeometry struct.

```
double del, az ;
TTTime result ;
result.deriv[0]=result.deriv[1]=result.deriv[2]=0;
strncpy (result.phase, phase_code, TTPHASE_SIZE );
dist(rad(geometry->source.lat),
     rad(geometry->source.lon),
     rad(geometry->receiver.lat),
     rad(geometry->receiver.lon),
     &del, &az ) ;
```

The results are returned in TTTime structures in the list *times. A return code indicates problems of different sorts.

```
if ( strcmp(phase_code, "P") == 0 ) {
    result.value = del / rad(p_velocity) ;
    pushtbl(*times, &result) ;
} else if ( strcmp(phase_code, "S") == 0 ) {
    result.value = del / rad(s_velocity) ;
    pushtbl(*times, &result) ;
} else {
    elog_log ( 0,
              "Bogus: unrecognized phase: '%s'\n",
              phase_code ) ;
    retcode = -1 ;
}
} else {
    elog_log ( 0, "Bogus: unrecognized model: '%s'\n",
              model ) ;
    retcode = -5 ;
}
return retcode ;
}
```

The Bogus_ucalc routine is substantially the same, but returns its results as TTSlow structures in a list *slows.

Extending dbe

dbe has several menus which can be extended by adding lines to the `.dbe.pf` parameter file. You may add options to the *Edit* menu, to the *Graphics* menu, and to the *Process* menu. In each case, you edit the portion of the parameter file which pertains to the schema you're using, and add a row or a list under edit, graphics or process.

Each row or element of the list identifies a base table for which the operation is valid, the label for the menu option, and finally the command line for the program which will execute the operation. This program must read an input view from stdin and perform the operation on each record in the view. For instance, the edit list under the schema `css3.0` looks like this:

```
edit          &Arr{
  instrument  Edit_text  db_textedit -
}
```

Because of this entry, dbe windows displaying the instrument table, or a view which includes the instrument table, have an option *Edit_text* under the Edit menu. Selecting some records from the window and then Edit->Edit_text brings up an editor on the corresponding response files.

`db_textedit` is just a shell script:

```
#!/bin/sh
FILES=`dbselect $1 'extfile()' | sort -u`
if [ ${XEDITOR-textedit} != "textedit" ] ; then
  $XEDITOR $FILES &
else if [ ${EDITOR-textedit} != "textedit" ] ; then
  xterm -e $EDITOR $FILES &
else
  for i in $FILES
  do
    /usr/openwin/bin/textedit $i &
  done
fi
fi
```

Most of it is spent deciding what editor to run; the kernel of the script is the line which runs `dbselect(1)` to get a list of external files, and `sort(1)` to eliminate duplicates.

Summary

Antelope provides a number of methods for customizing it to adapt to local practice, and apply the power of its programs to local situations. If you have a special waveform format, a particular method of calculation magnitudes, or your own location program, you can integrate these into the Antelope structure in a straightforward way.

Exercises

1. Run the Bogus travel time calculator using `ttcalc(1)` and compare the results with `tau-p` and `iasp-91`.
2. Write a `pf` function which evaluates a Datascope expression given the database table and an expression.
3. Design and create a packet to contain alert messages of some sort. Compare with using a `pf` packet for the same purpose.

Larger Development Efforts

When you have only one or two specially developed programs at your site, you can be rather cavalier about managing them. However, as you develop more programs, and more people are involved (in development or in using the programs), it becomes important to have an overall strategy and organization. This chapter outlines some methods for achieving this organization, and makes some suggestions for good standard practice. These suggestions are in large part the guidelines BRTT follows; they grew out of considerable development experience and sometimes painful trial and error.

First of all, it's important to have a central repository for source code. This can be as simple as a directory hierarchy which is accessible by all developers. It's best that this hierarchy be kept somewhere other than someone's home directory however.

It's very useful to keep the source code in a source code control system. This makes it possible to track versions and changes and makes it somewhat to document the reasons for changes. Solaris packages SCCS as part of the compiler tools in `/usr/ccs`. This system is quite workable. At BRTT, however, we use CVS. This is a very popular choice among developers today. (It is also, not by accident, the choice made by the Antelope contrib repository.)

CVS allows keeping an entire hierarchy of source directories in a central repository. Each developer checks out their own copy of this source tree and pursues develop-

ment in their own world. When a developer is ready, they put their changes back into the central repository. Any conflicts (i.e., if two people have changed the same source code) are resolved by hand editing at this point.

cvs example

You may need to obtain `cvs` from its current home <http://www.cvshome.org>, and install it. Then you begin by making a repository. Since this will contain your source code and all its history, choose a disk and machine on which you can rely (and consider your backup strategy).

There are lots of commands, but you typically need only two; try `cvs --help-commands` for a list.

```
% cd /your/directory
% cvs -d $cwd/cvs init
% ls cvs
CVSROOT/
```

It's probably useful to set an environment variable `CVSHOME` to `/your/directory` in your `.tcshrc` file

Now `cd` to the directory where your source code resides. You may want to do a `make clean` to remove old executable and object files, and get rid of any other dross which has accumulated. The process of importing a source tree tries to eliminate object files and the like, but it's more reliable to eliminate things directly. Enter a command like this:

```
% cvs import new-cvs-dir your_name r0
```

This imports most of the contents of the current directory and all directories below it into the `cvs` repository you just created, under a new directory there named `new-cvs-dir`. Probably you'll want to choose a better name. The third argument is supposed to be a source, and the fourth a version; these arguments don't seem to be of much use later, but `cvs` is very picky and doesn't allow dashes or periods, for instance.

Now, you have the repository. Very likely, it has all the source files you wanted in it. You or anyone with write access to the repository can checkout the source code:

```
% cvs checkout new-cvs-dir
```

This new copy of the source is your playpen; edit as you like without affecting the repository or other users. When you have something that is working check the changes back in (from your working directory) with the command:

```
% cvs commit
```

This command figures out what files have changed, and prompts you to enter an explanation, which is stored with every changed file. If you've added files, you must let cvs know before the commit:

```
% cvs add new1.c new.h
```

A commit adds these to the repository, too. Adding a directory is only slightly harder; first use `cvs add newdir` to add the new directory, then add the files in the new directory.

Now, the commit may fail if someone else has also changed a file which you've changed. In this case, you must update your copy of the file to the current repository version, edit in your changes, and then commit. To update, use the update command:

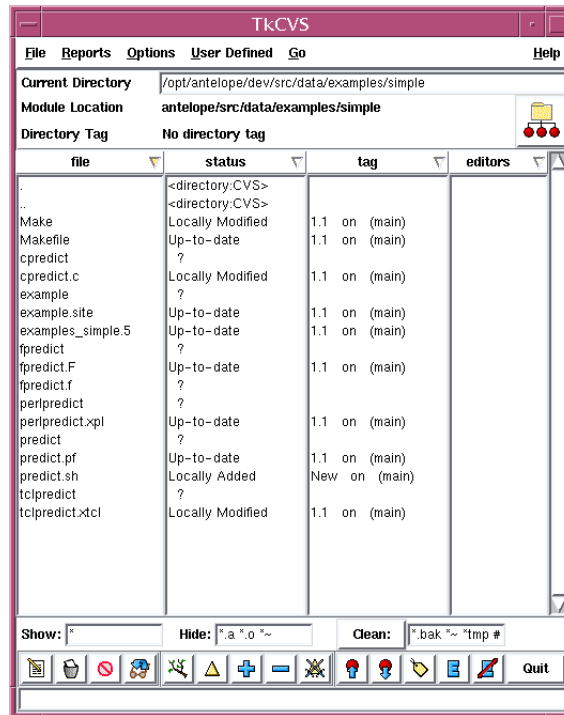
```
% cvs update
```

cvs will save a copy of your version in a dot file, something like `.#yourfile.c`, and then attempt to merge the repository version and your version. The result is not always illuminating. You may find it's easier to remove it, use `cvs update` to get an unblemished copy from the repository and use `tkdiff(1)` or `sdiff(1)` to merge the differences between `.#yourfile.c` and the `yourfile.c` from the repository.

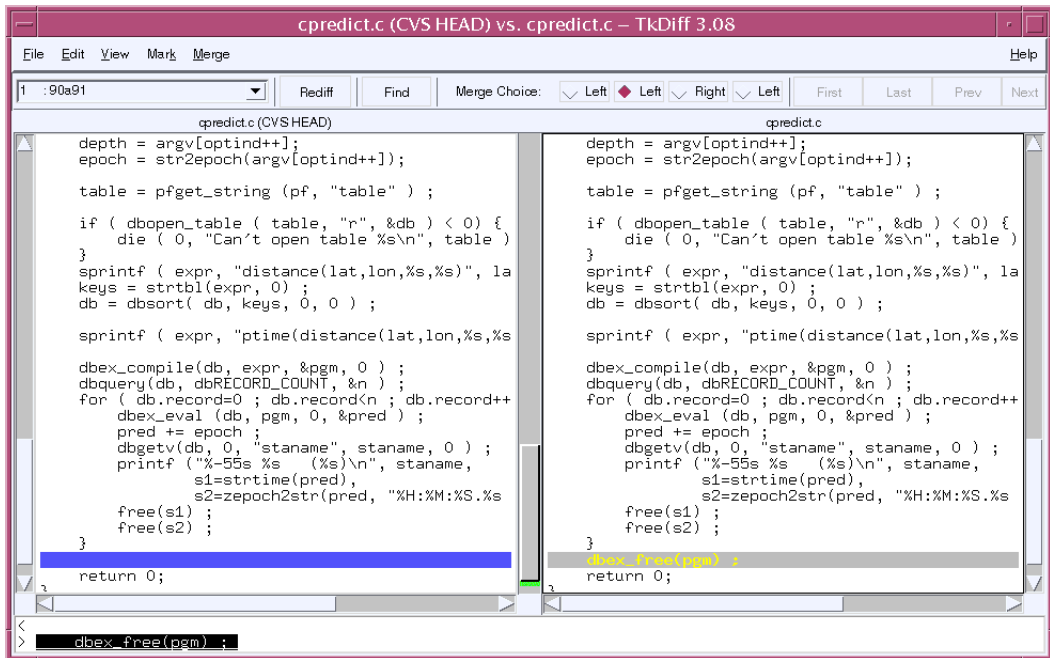
A source repository is useful because

- multiple people can get copies and work independently
- you keep track of changes and can figure out where something went bad
- if something goes bad, you can back out to a working system
- it's one more level of backup

You may find the tool `tkcvs(1)` useful in exploring your cvs repository, and looking up change histories and log files.



Even more useful is a program that comes with `tkcvs` (but can be found separately), `tkdiff`. It's a diff program which is similar to `sdiff`, but uses an X-window to graphically display differences. It's far easier to appreciate the context and breadth of the changes this way.



Coordinated Makes

It's very useful to organize the source so that a single make at the top level causes make to run in every sub-directory. This is easily arranged if you use \$ANTELOPE-MAKE. In directories which are empty except for subdirectories, create a Makefile with the single statement;

```
include $(ANTELOPEMAKE)
```

Running make in this directory will cause make to be executed in each subdirectory (in alphabetical order). If necessary, you can override the default and specify the directories in the order in which they're to be entered. After the include line, add a line defining DIRS:

```
include $(ANTELOPEMAKE)
DIRS=first second third
```

(Of course, when directories are added or removed, this line may need to be changed). To suppress all descent into subdirectories, define `DIRS` as empty:

```
include $(ANTELOPEMAKE)
DIRS=
```

Installation Directories

Normally, an Antelope Makefile installs directly into the `$ANTELOPE` hierarchy. You might prefer to keep locally generated programs separate. In that case, you have two easy alternatives:

1. Add the line `SUBDIR=/mydir` before the include line. This will cause programs, man pages and data files to be installed into a new hierarchy `$ANTELOPE/mydir`.

```
SUBDIR=/mydir
include $(ANTELOPEMAKE)
```
2. Add the line `DEST=/yourdest` before the include line. This will cause programs and other files to be installed into a hierarchy under `/yourdest`.

```
DEST=/yourdest
include $(ANTELOPEMAKE)
```

Either of these solves the problem of installation. However, you must in addition add this new bin directory to your `PATH`, add the new man directory to your `MANPATH`, and set the environment variable `DATAPATH` appropriately.

Exercises

1. Collect the important programs written at your location into one single source hierarchy.
2. Arrange so that a single make command at the top of the hierarchy makes all the programs.
3. Enter all the source files into CVS.
4. If you wish, change the Makefiles so that `make install` puts the binaries not into `$ANTELOPE/bin`, but somewhere else.

Summary

The intent of this document is to introduce the various facilities, libraries and tools which are a part of Antelope, and also to mention a variety of tools which are not part of Antelope, but are very useful to us. Hopefully, you have gotten a start on solving your problems using the Antelope toolkit.

Of course, the information is incomplete. You must go to the man pages to get detailed information. If you think we've left out important things or made mistakes in this document (or any other document, for instance the man pages), we want to hear about it.

There is an Antelope community that is interested in your work, and can be helpful if you have problems. There is an Antelope User Group web site at

`http://www.indiana.edu/~aug/`

At this site, you can find source code for a wide variety of programs using Antelope libraries, some web based documentation for certain programs, and a bulletin board for web based discussion.

You can also post questions and advice to the antelope users mailing list by sending email to

`antelope-users@brtt.com`

You must first join the list, which you can do by sending email to

`antelope-users-request@brtt.com`

with a single line in the body of the email, *subscribe*.

In addition there is a collaborative FAQ at

`http://www.brtt.com/faq`

Please post to the mailing list, or add to the faq.

Exercises

1. The next time you can't figure out something related to Antelope, post a question to `antelope-users@brtt.com`.
2. When you get a reasonable answer, put both question and answer into the FAQ at `http://www.brtt.com/faq`.
3. If you write a program other people might use, gain fame if not fortune by contributing it and its man page to the user group archive.

Here is a brief list of books about programming which you may find enlightening and sometimes entertaining.

Introduction to Unix

- *The UNIX Programming Environment*, Brian W. Kernighan, Rob Pike
This is the classic book on programming in Unix, by some of the original authors of Unix.
- *Learning the Unix Operating System*, Jerry Peek, Grace Todino, John Strang
- *UNIX in a Nutshell: System V Edition*, Arnold Robbins
- *Unix Power Tools*, Jerry Peek, Tim O'Reilly, Mike Loukides
This collection of questions, answers and examples is useful for even the experienced Unix user.

System administration

- *When You Can't Find Your UNIX System Administrator*, Linda Mui
- *UNIX System Administration Handbook*, Evi Nemeth, et al

Perl

- *Programming Perl*, Larry Wall, Tom Christiansen, Jon Orwant
You'll probably want this and the pocket reference below if you program in Perl.
- *Perl 5 Pocket Reference, Programming Tools*, Johan Vromans
- *Perl/Tk Pocket Reference*, Stephen Lidie
For a long time, the only useful documentation for the Tk interface in Perl.
- *Mastering Perl/Tk, Graphical User Interfaces in Perl*, Nancy Walsh, Stephen Lidie
- *perltidy*
<http://perltidy.sourceforge.net>

C

- *The C Programming Language*, Brian W. Kernighan, Dennis M. Ritchie
Another classic
- *The Standard C Library*, P. J. Plauger
- *Expert C Programming (Deep C Secrets)*, Peter van der Linden
Amusing and interesting insights about C
- *Advanced Programming in the UNIX(R) Environment*, W. Richard Stevens
A lot of detailed examples of using C library calls.
- *POSIX Programmers's Guide*, Donald Lewine
- *POSIX.4*, Bill O. Gallmeister
If you have a choice, stick to POSIX standard (and ANSI C standard) usage.
- *Annotated ANSI C Standard*, Herbert Schildt
very useful because it explains what the standard means.
- *Threads Primer*, Bil Lewis, Daniel J. Berg
a mercifully short and concise book.

Tcl/Tk

- *Tcl and the Tk Toolkit*, John K. Ousterhout
by the author of Tcl/Tk
- *Practical Programming in Tcl and Tk*, Brent B. Welch
- *Tcl/Tk Pocket Reference*, Paul Raines

CVS

- *Cvs Pocket Reference*, Gregor N. Purdy
- CVS repository
<http://www.cvshome.org/>
- CVS documentation
<http://www.cvshome.org/docs/manual/cvs.html>
- tkcvs
<http://www.twobarleycorns.net/tkcvs.html>

Miscellaneous

- ups home
<http://www.concerto.demon.co.uk/UPS/>
- Purify
http://www.rational.com/products/purify_unix/index.jsp
- Virtual Network Computing (Vnc)
<http://www.uk.research.att.com/vnc/>

Antelope

- local documentation at your site.
<file:/opt/antelope/4.4/antelope.html>
- BRTT home page
<http://www.brtt.com>

- Antelope User Group Web Site
<http://www.indiana.edu/~aug/>
- Antelope collaborative FAQ
<http://www.brtt.com/faq>

Fun

- *The Practice of Programming*, Brian W. Kernighan, Rob Pike
- *Elements of Programming Style*, Brian W. Kernighan, P. J. Plauger
Good advice about how to program
- *The Psychology of Computer Programming*, Gerald M. Weinberg
A funny and interesting classic, though overpriced in it's current edition.
Maybe you can find it in the library.
- *The Mythical Man-Month: Essays on Software Engineering*, Frederick P. Brooks

Exercises

1. Find your own references at these websites:
 - Unix Reference Desk
<http://www.geek-girl.com/unix.html>
 - Amazon Books
<http://www.amazon.com>
 - Google
<http://www.google.com>