ColdSync Conduits

Specification and Hacker's Guide
Version 2.2.5

Andrew Arensburger

For external use only. Do not attempt to read this manual while operating a motor vehicle.

Introduction

This manual is intended as a reference. Each chapter and, to the extent possible, each section, is intended to stand on its own. Consequently, if you read it from beginning to end, expect a certain amount of repetition.

The first chapter, "Overview," is for the general user. It explains what conduits are, why you would want to use one, and how to set Coldsync up to use them.

The second chapter, "Tutorial," is a quick-start guide for those who want to write their own conduits. It presents a short but functional conduit in Perl, with annotations and comments, as well as a set of recommendations for conduit authors.

The third chapter, "Specification," formally describes interaction between ColdSync and conduits, what conduits are allowed to do, and what they are forbidden from doing. This chapter is no substitute for the source, but is probably easier to read.

1 Overview

By default, ColdSync is simply a fancy way of keeping a backup of what's on your Palm. This is all well and good, but what you'd really like is for your Palm to share its information with the other tools you have on your desktop.

That's where conduits come in. A conduit is an external program that ColdSync runs at certain times; you can use a conduit to extend ColdSync's behavior.

ColdSync is a generalist: it doesn't know a Calendar from a MineSweeper, so it has to treat all applications and their data in a fairly generic manner. Conduits, then, are specialists: they usually know all about one particular type of application and nothing else. This allows them to do the Right Thing for that particular application.

1.1 Kinds of Databases

Since Palms don't have disks, everything has to be in memory. Consequently, PalmOS doesn't distinguish between RAM and files, the way other operating systems do. Instead, everything is a database.

There are two types of database: *record databases* and *resource databases*.

Record databases consist, naturally, of records. These are the databases that hold all of your information. The "AddressDB" database, for instance, has one record for each address in the Address Book application.

Resource databases are a bit more organized: each entry has a four-letter type, and a numerical identifier. Most resource databases are applications, containing one or more '`code`' entries, perhaps a few '`Tbmp`' (bitmap image) entries, and so forth. The types are documented strings that indicate what the resource is: '`code`' resources contain executable code, '`Tbmp`' resources are always bitmap images, and so forth. The numerical identifiers serve to tell resources of the same type apart.

Not coincidentally, resource databases are very similar to the resource fork in MacOS files.

Note that, for technical reasons, resource datatabases do not lend themselves well to syncing, and ColdSync pretty much ignores them.

1.2 Creators and Types

Each database also has a four-letter *type* and a four-letter *creator*. Each PalmOS application has a unique creator string. Every database that it creates has the the same creator string as the application. For example, the built-in Datebook application has the creator '`date`'. The record database that contains the Datebook entries also has the creator '`date`'.

An application can potentially create several databases to represent different kinds of data. The built-in To Do application creates three separate record databases: '`ExpenseDB`', with creator `exps` and type `DATA`; '`CitiesDB`', with creator `exps` and type `city`; '`VendorsDB`', with creator `exps` and type `vend`. All of these databases have the same creator field as the application that created them, but have different types, since they contain different kinds of information.

Since each conduit typically handles only one specific kind of database, you have to tell ColdSync what it is. For instance, to run the '`kab-fetch`' conduit on databases with creator '`addr`' and type '`DATA`', you would add the following to your '`.coldsyncrc`' file:

```
conduit fetch {
        path: "/usr/local/libexec/coldsync/kab-fetch";
        type: addr/DATA;
}
```

An asterisk in a creator or type specification acts as a wildcard:

```
type: addr/*;
```

the conduit will be run for every database with creator '`addr`', regardless of type. Likewise:

```
type: */DATA;
```

the conduit will be run for every database with type '`DATA`', regardless of creator (though this isn't usually very useful). Finally, you can specify wildcards for both the creator and type:

```
type: */*;
```

In this case, the conduit will be run for every database.

1.3 Conduit Flavors

ColdSync defines several *flavors* of conduits. Each flavor performs a different function and is designed for a different purpose. Currently, four flavors are implemented.

Sync conduits implement the synchronization between a database on the Palm and its backup copy on the desktop, what is referred to here as the main sync.

The generic default conduit is a Sync conduit. It reads records from the Palm, compares them to what's on the desktop, and uploads or downloads records as necessary to ensure that both the Palm and the desktop have identical, up-to-date data.

Since Sync conduits can talk directly to the Palm, they are the most powerful, but also the most complex type of conduit. In the most common case, where all you want to do is synchronize data between the Palm and a file or database, it is simpler to write a Fetch conduit and a Dump conduit. In some cases, however, this is not possible: if you want to set the time on the Palm, your conduit needs to do so while the Palm is connected. For this, you would need to use a Sync conduit.

Fetch conduits run before the main sync. The idea is that a Fetch conduit will modify its database before ColdSync compares it to the version on the Palm. For instance, a conduit might read a list of scheduled meetings from '`/usr/local/libdata/meetings`' and make sure that they're all in the Address Book database in your backup directory. Then ColdSync wil make sure that any changes to the meeting list will be reflected on your Palm.

To specify a Fetch conduit in your '`.coldsyncrc`', use `fetch` or `pre-fetch` as the flavor argument to `conduit`, e.g.:

```
conduit fetch {
        path: "/usr/local/libexec/coldsync/get-meetings";
        type: date/DATA;
}
```

If you specify several conduits of the same flavor for the same database, they will be run one at a time, in the order in which they appear in '.coldsyncrc'. This allows you to chain the effects of several conduits. Bear in mind, however, that they may interfere with each other

Dump conduits run after the main sync. The intended purpose of a Dump conduit is to read its database after ColdSync is done with it, and copy its contents to some other file, in some other format. For instance, if you have made changes to your address list on the Palm, ColdSync will make sure that the backup copy in your home directory contains the same information as the Address Book database on the Palm. After that's done, a conduit can export this list in your favorite address book application's format.

To specify a Dump conduit in your '.coldsyncrc', use `dump` or `post-dump` as the flavor argument to `conduit`, e.g.:

```
conduit dump {
        path: "/usr/local/libexec/coldsync/save-meetings";
        type: date/DATA;
}
```

If you only have a Fetch conduit defined for a database, it can implement "Desktop overwrites handheld" syncing, where the master copy is kept on the desktop machine, and any changes made on the Palm are lost.

Conversely, if you only have a Dump conduit defined for a database, it can implement "Handheld overwrites desktop" syncing, where the master copy is kept on the Palm, and any changes made on the desktop machine are lost.

If you have both a Fetch and a Dump conduit defined for a database, they can implement two-way syncing, where changes made on the desktop are propagated to the Palm and vice versa. Since the logic of syncing can get rather hairy, especially in difficult cases, it's easier to write two simple conduits, a Fetch and a Dump, than it is to write a single conduit that does a two-way sync.

Install conduits provide a hook for manipulating databases before they're installed. In the installation phase, ColdSync looks for databases in the install directory ('~/.palm/install' by default) and runs any applicable Install conduits on them. After that, it installs any remaining files in the install directory.

If you don't like the built-in Datebook application, but people give you databases in this format, you could use an Install conduit to convert them to your favorite format. Or, depending on how bleak the future becomes, you may find it necessary to have an Install conduit check new databases for viruses, and delete them if they're infected.

1.4 Preferences

The Palm has two special databases, "Saved Preferences" and "Unsaved Preferences," that contain preferences. Preferences are the PalmOS equivalent of dot files: they are neither applications nor data, but rather user preferences that affect how an application runs. The signature for outgoing mail in the Mail application is a preference. So is the "Owned by" text you can set in the Prefs application.

ColdSync conduits can request to be given the value of a preference and act accordingly. For instance, a mail conduit might request the signature preference and append it to outgoing messages.

There are two types of preferences: saved and unsaved. I have no idea what the difference is between these two types. Perhaps saved preferences are those that have found God.

2 Tutorial

This chapter shows, by example, how to write ColdSync conduits. The examples are written in Perl, simply because I happen to like it. However, you can use any language you like to write conduits.

The example conduits in this chapter use the `ColdSync.pm` module that's part of the ColdSync distribution, and also the `p5-Palm` module from `http://www.ooblick.com/software/coldsync/`.

2.1 Conduit Workings: A Quick Overview

A conduit is simply a program, one that follows the ColdSync conduit protocol (see Chapter 3 [Specification], page 19).

In a nutshell, ColdSync runs a conduit with two command-line arguments: the string `conduit`, and another that indicates the conduit flavor, either `fetch` or `dump`.

ColdSync then writes a set of header lines to the conduit's standard input, e.g.,

```
Daemon: coldsync
Version: 2.2.5
InputDB: /homes/arensb/.palm/backup/ToDoDB.pdb
Phase-of-the-Moon: lunar eclipse
```

followed by a blank line.

The conduit reports its status back to ColdSync by writing to standard output, e.g.:

```
202 Success.
```

The three-digit code indicates whether this is an error message, a warning, or an informational message. See Section 3.5 [Status Codes], page 24. The rest of the line is a text message to go with the status code. It is not parsed by ColdSync; it is intended for human readers.

A conduit should print such a message before exiting, to indicate whether it was successful or not.

2.2 `todo-dump`

Let's write a Dump conduit that writes the current To Do list to a file. This is a single-flavor conduit, so we'll use the following template:

```
#!/usr/bin/perl
use Palm::ToDo;
use ColdSync;

# Declarations and such go here.
```

```
StartConduit("dump");

# Actual conduit code goes here

EndConduit;
```

The `Palm::ToDo` module is a parser for ToDo databases; it adds hooks so that when the conduit reads the ToDo database, its records will be parsed into structures that can easily be manipulated by a Perl program (see Palm::ToDo(1)).

The `ColdSync` module provides a framework for writing conduits, and defines the `StartConduit` and `EndConduit` functions.

`StartConduit` takes one option indicating the conduit flavor (Dump, in this case). It checks the command-line options and makes sure the conduit was invoked with the proper flavor. It reads the headers from standard input and puts them in `%HEADERS`. If the conduit was given a `InputDB` header, `StartConduit` loads the database into `$PDB`.

`EndConduit` takes care of cleaning up when the conduit finishes. For Fetch conduits, it writes `$PDB` to the file given by `$HEADERS{OutputDB}`.

Starting with this template, all we need to do now is to insert the actual code:

```
#!/usr/bin/perl
use Palm::ToDo;
use ColdSync;

$OUTFILE = "$ENV{HOME}/TODO";           # Where to write the output

format TODO =
@ @ @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$marker, $priority, $description
      ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
        $note
.

StartConduit("dump");

open OUTFILE, "> $OUTFILE" or die("401 Can't open $OUTFILE: $!\n");
select OUTFILE;
$~ = TODO;                      # Set the output format

foreach $record (@{$PDB->{"records"}})
{
        $marker         = ($record->{"completed"} ? "x" : "-");
        $priority       = $record->{"priority"};
        $description    = $record->{"description"};
        $note           = $record->{"note"};

        write;
}

close OUTFILE;
```

```
EndConduit;
```

The ColdSync.pm module provides wrappers for Perl's die and warn functions, so that their messages will be passed back to ColdSync. The rest of the code should be self-explanatory.

2.3 pine-aliases

Now that we've seen a trivial conduit, let's take a look at a slightly more complicated one: a conduit to synchronize addresses in the Palm Address Book database with those in Pine's address book.

Note that this is still just a tutorial conduit: we'll be making some simplifying assumptions that will make this conduit unsuitable for use in the real world.

Having said this, let's take a look at the conduit:

```
#!/usr/bin/perl
use Palm::Address;
use ColdSync;

$PINE_ALIASES = "$ENV{HOME}/.addressbook";

ConduitMain(
        "fetch" => \&DoFetch,
        "dump"  => \&DoDump,
        );
```

Unlike todo-dump (see Section 2.2 [todo-dump], page 7), pine-aliases is a multi-flavor conduit: it can be used either as a Fetch conduit or as a Dump conduit. For this reason, we use ConduitMain rather than StartConduit/EndConduit.

There are several reasons why one might want to write a multi-flavor conduit like this one. The first is that the Fetch and Dump functions really just implement the two halves of a single conduit that performs two-way synchronization between the Palm and Pine.

Secondly, we'll be writing some convenience functions that will be used by both &DoFetch and &DoDump, so it makes sense to keep them together.

Finally, in many cases, the two things that one is synchronizing (in this case the Palm Address Book and Pine's addressbook file) don't contain the same information, or represent it in such a way that it's difficult to convert one to the other, and the conduit writer must resort to a number of tricks to perform the sync correctly.

For instance, the Fetch conduit for kab tries to save each person's fax number in the Palm database. If there is no fax field, it will append "(212) 123-4567 (fax)" to the "Other" field. Therefore, the kab Dump conduit must look for the fax number in the "Other" field as well as the "Fax" field. Keeping the two conduits together in the same file makes it easier to keep track of these sorts of tricks and make sure that the two conduits work properly.

ConduitMain takes as its arguments a table that tells which function to call for each flavor. When the conduit is run, ConduitMain parses and checks the command-line arguments, reads the headers from standard input and stores them in the hash %HEADERS, and calls the appropriate function. If an `InputDB` header was specified, that file will be read into the variable `$PDB`. Then it calls the flavor-specific function (in this case, either `&DoFetch` or `&DoDump`) to do the actual work of the conduit, and finally cleans up: for Fetch conduits, it writes the contents of `$PDB` to the file specified by the `OutputDB` header.

2.3.1 &DoFetch

The `&DoFetch` function reads Pine's alias file. For each address that it finds there, it updates the email address in the appropriate record in the Palm database.

```
sub DoFetch
{
        my %aliases = ();

        if (!defined($PDB))
        {
                $PDB = new Palm::PDB;
                $PDB->Load($HEADERS{"OutputDB"}) or
                        die "502 No input database\n";
        }

        open ALIASES, "< $PINE_ALIASES" or
                die "Can't open $PINE_ALIASES: $!\n";

        while (<ALIASES>)
        {
                my $alias;
                my $addr;
                my $fullname;
                my @rest;

                chomp;
                ($alias, $fullname, $addr, @rest) = split /\t/;
                $aliases{$fullname} = $addr;
        }

        my $fullname;
        my $address;

        while (($fullname, $address) = each %aliases)
        {
                my $record = &find_person($PDB, $fullname);

                next if !defined($record);      # No entry in PDB

                my $pdb_addr = &get_address($record);

                next if $pdb_addr eq $address;
                                # It already matches. Ignore it.
```

```
                        print STDOUT "101 Setting $fullname -> $address\n";
                        &set_address($record, $address);
                }
                close ALIASES;

                return 1;               # Success
        }
```

The `InputDB` header is optional for Fetch conduits, so `$PDB` may not have been initialized. But `pine-aliases` does not create a new database from scratch; it only modifies an existing one. If no `InputDB` database was specified, we load the database specified by `OutputDB`.

The body of `&DoFetch` is divided into two phases: in the first phase, it reads the Pine alias file and builds a hash, `%aliases`, that maps each full name to its email address. The second phase goes through this map and updates each record in `$PDB`. This two-phase approach may seem overly complex; the reasons for it are discussd in Section 2.3.4 [pine-aliases Limitations], page 16.

Each line in the Pine address book contains a set of tab-separated fields: the person's alias, full name, email address, and a few others that we don't use.

We'll need some way of figuring out which Pine alias goes with which Palm Address Book record. Since the Pine alias file does not list Palm record IDs and Palm records don't list mail aliases, we'll settle on the full name as the next best way of uniquely identifying a person.

The second phase of `&DoFetch` uses a number of helper functions: *&find_person* takes a person's full name and returns a reference to the corresponding record in `$PDB`; `&get_address` extracts the email address from that record; and `&set_address` sets the email address in the record.

One important thing to note is that `&set_address` marks the record as dirty. During a normal sync, ColdSync only considers those records that have changed in some way. When we update the address, we need to make sure that the record is marked as dirty; otherwise it will not be uploaded to the Palm.

When `&DoFetch` returns, `ConduitMain` writes `$PDB` to the file given by `$HEADERS{"OutputDB"}` and exits. Then, during the main sync, ColdSync will upload to the Palm any records `pine-aliases` has modified.

2.3.2 `&DoDump`

The `&DoDump` function implements the Dump conduit:

```
        sub DoDump
        {
                open ALIASES, "< $PINE_ALIASES" or
                        die "502 Can't read $PINE_ALIASES: $!\n";
                open ALIASES_NEW, "> $PINE_ALIASES.new" or
                        die "502 Can't write $PINE_ALIASES.new: $!\n";

                while (<ALIASES>)
```

```
{
        chomp;

        my $alias;
        my $addr;
        my $fullname;
        my @rest;
        my $record;

        ($alias, $fullname, $addr, @rest) = split /\t/;

        $record = &find_person($PDB, $fullname);
        if (!defined($record))
        {
                # This name doesn't appear in $PDB.
                print ALIASES_NEW $_, "\n";
                next;
        }

        # This person appears in both the alias file and in
        # the PDB.
        my $pdb_addr = &get_address($record);

        if (defined($pdb_addr))
        {
                # Found an address
                print STDOUT "101 $fullname -> $pdb_addr\n"
                        if $pdb_addr ne $addr;
                print ALIASES_NEW
                        join("\t", $alias, $fullname,
                                $pdb_addr, @rest),
                        "\n";
                next;
        }

        # The PDB record doesn't have an email address. Mark it
        # as deleted.
        my $year;
        my $month;
        my $day;

        ($year, $month, $day) = (localtime)[5,4,3];
        $year %= 100;
        $month++;

        $alias = sprintf "#DELETED-%02d/%02d/%02d#%s",
                        $year, $month, $day, $alias;
```

```
                print ALIASES_NEW
                        join("\t", $alias, $fullname, $addr, @rest),
                        "\n";
        }

        close ALIASES_NEW;
        close ALIASES;
        rename "$PINE_ALIASES.new", $PINE_ALIASES or
                die "Can't rename $PINE_ALIASES.new: $!\n";

        return 1;               # Success
}
```

In `&DoDump`, we read each line of '`~/.addressbook`' in turn and write a possibly-update version to '`~/.addressbook.new`'. The reasons for using two files is twofold: first of all, the length of a line might change, so we can't just update the file in place. Secondly, if anything goes wrong during the sync, we can simply abort before moving the new file into place, and leave the old alias file untouched, rather than risk corrupting it.

Again, we use `&find_person` to look up the Palm record corresponding to a person's full name, and `&get_address` to extract the email address from the record. There are three cases we need to consider:

- There is no record corresponding to this alias.

  There are two approaches we can take here: we can either delete the Pine alias (simply by not writing it to `ALIASES_NEW`), or we can ignore it. Since we're not trying to make sure that every Palm record has a corresponding Pine alias, we'll take the latter approach.

- There is a record, and it has an email address.

  We write the alias to `ALIASES_NEW`, with the email address listed in the Palm record. This may or may not be different from what was there before, but it doesn't matter: this is the most up-to-date address.

- There is a record, but it doesn't have an email address.

  In this case, we'll assume that the email address was deleted on the Palm, otherwise the Fetch conduit would have uploaded the email address. Hence, this email address is obsolete and should be commented out. In general, it is preferable to comment things out rather than delete them: that way, if there's a bug somewhere, the information isn't permanently lost.

2.3.3 Helper functions

These are the helper functions used in `pine-aliases`.

`&find_person` takes a reference to a `Palm::Address` and a full name, and returns a reference to the record corresponding to that name:

```
sub find_person
{
        my $PDB = shift;
        my $fullname = shift;
```

```
        my $record;

        foreach $record (@{$PDB->{"records"}})
        {
                next unless ($record->{"fields"}{"firstName"} . " " .
                        $record->{"fields"}{"name"}) eq $fullname;
                return $record;
        }
        return undef;           # Failure
}
```

Since Palm Address Book records don't contain a full name field, we construct one from the first and last names, and see if it matches.

Note that a better version of this function would also consider other fields: an entry such as "Ooblick Technical Support" might be listed on the Palm with no first or last name, but with the company field set to "Ooblick" and the title field set to "Technical Support".

`&get_address` takes a reference to a Palm record, and extracts the email address, if any:

```
sub get_address
{
        my $record = shift;
        my $field;

        # Look through all of the "phone*" fields
        foreach $field ( qw( phone1 phone2 phone3 phone4 phone5 ) )
        {
                next unless $record->{"phoneLabel"}{$field} == 4;

                # Found the (or an) email field
                my $addr = $record->{"fields"}{$field};

                $addr =~ s/\n.*//;      # Keep only first line

                # Remove parenthesized expressions
                $addr =~ s/\([^\)]*\)//;
                $addr =~ s/^\s+//;      # Remove leading whitespace
                $addr =~ s/\s+$//;      # and trailing whitespace

                return $addr;
        }

        return undef;           # Couldn't find anything
}
```

This was made into a separate function for clarity: the Palm Address Book record format does not contain a separate field for the email address. Rather, it has five fields named `phone1` through `phone5`, each of which can be a home phone, work phone, fax number, email address, etc. See Palm::Address(1) for details.

`&get_address` looks at each phone field in turn until it finds one whose `phoneLabel` is 4, meaning "Email". It extracts the useful part of the address and returns it.

Note that this function is very simplistic: all it does is remove the parentheses from addresses of the form

```
JDoe@ooblick.com (John Doe)
```

The general case is much more complex.

`&set_address` is the converse of `&get_address`: it stores an email address in a record:

```
sub set_address
{
        my $record = shift;
        my $addr = shift;
        my $field;

        # Find the Email phone field
        foreach $field ( qw( phone1 phone2 phone3 phone4 phone5 ) )
        {
                next unless $record->{"phoneLabel"}{$field} == 4;

                # Found it.
                $record->{"fields"}{$field} = $addr;
                $record->{"attributes"}{"dirty"} = 1;
                return;
        }

        # No Email field found.
        foreach $field ( qw( phone1 phone2 phone3 phone4 phone5 ) )
        {
                next if $record->{"phoneLabel"}{$field} =~ /\S/;

                # Found an empty field
                $record->{"phoneLabel"}{$field} = 4;
                $record->{"labels"}{$field} = $addr;
                $record->{"attributes"}{"dirty"} = 1;
                return;
        }

        # No Email fields, and no empty fields. Fail silently.
        return;
}
```

Again, due to the format of Palm Address Book records, this function is more complicated than it seems that it ought to be.

In the simplest case, we look at all of the phone fields, find one marked “Email”, and update it.

If there is no email field, `&set_address` tries to find an empty field and turn it into an email field, then writes the address to that field.

If there are no empty fields, we'll give up, since this is just a tutorial. A real conduit ought to keep trying: it might consider adding the email address to the “Other” phone field, if there is one. As a last resort, it might add the email address to the note. Of course,

`&get_address` also needs to know about all of the places where an email address might lurk.

In any case, `&set_address` marks the record as being dirty, so that it will be uploaded to the Palm at the next sync.

2.3.4 Limitations of `pine-aliases`

The conduit we've just seen is just a tutorial. For the sake of simplicity, we've ignored several real-world considerations that would have made the code even harder to read.

The first simplifying assumption we've made is that there is only one email address per person. In the real world, people often have a home address and a work address. To deal with this, `&DoFetch` should collect an array of addresses for each person, then make sure that each address in the array exists in the Palm record (this is why `&DoFetch` is split up into two phases).

One issue that complicates matters is that a Palm Address Book record might contain multiple phone fields marked "Email". `&get_address` ought to handle this case. The other side of the issue is that `&set_address` shouldn't just dump all of the email addresses into the first "Email" phone record that it finds, otherwise the second and subsequent addresses will be duplicated.

Secondly, we've assumed that each full name uniquely identifies a single person. This obviously fails if the user knows two people named John Smith. In the case of `pine-aliases`, we can get away with documenting this limitation and requiring the user to list one of them as "John Allan Smith" and the other as "John Paul Smith". We might also consider setting up a separate file that maps Pine mail aliases to Palm record IDs, since those are unique identifiers in their respective domains.

Finally, `&set_address` shouldn't fail so easily: if it fails to add an email address to the record, then at the next sync, the corresponding Pine alias will be commented out. If a record is so full that there are no empty phone records, then obviously it's very important, and the user would be rather upset at losing this email address.

2.4 Style and Warnings: things to watch out for

The conduit presented above is very simple, and does not address many problems you will run into when writing "real" conduits.

- When writing a Fetch conduit, think about what it should do if there is no input: if the input file is empty, this might mean that there are no records, and the conduit should write a database with no records.

  However, if the input file doesn't exist, then it's probably a bad idea to delete all of the records in the backup database. In this case, it's probably best just to abort: most likely, the file was accidentally deleted, or else it's on an NFS partition and the remote host is down, or the file is supposed to be generated by a Dump conduit that hasn't been run yet.
- Deleting records: the obvious way to delete a record is to simply fail to write it to the output database. This is a bad idea, because of the way that fast syncs work. Use `$pdb->delete_Record($record)` or `$pdb->delete_Record($record, 1)`instead

When you modify a record on the Palm, the record is marked as being dirty (modified). Likewise, when you delete a record on the Palm, it is not actually deleted; rather, it is simply marked as being deleted.[1] This way, ColdSync needn't bother downloading the entire database from the Palm to see which records have changed: it simply asks the Palm for a list of records that have been modified and/or deleted.

ColdSync does the same thing in reverse, as well: it reads its backup copy of the database, looking for records that are marked as having been modified or deleted. It uploads modified records, and tells the Palm to purge the deleted records.

If you're writing to the backup file and simply fail to write a deleted record, ColdSync will never notice this record and won't tell the Palm to delete it. It will remain on the Palm, and you will have to delete it manually.

- If you're converting a Palm database to another format, you'll often find that the Palm database and the other format hold different information. For instance, the KDE address book, `kab`, does not distinguish between home and work telephone numbers. On the other hand, it allows you to specify a person's URL, which the Palm Address Book does not directly support.

  Keep these sorts of differences in mind, or you'll risk losing information. One approach is to start by reading both the source and destination files, modify the records as necessary, and then write the resulting file. That way, if the output file has a field that doesn't correspond to anything on the Palm (like URLs in `kab` files, above), you won't delete those fields.

  An additional benefit of this approach is that if you encounter a fatal error in the middle of processing, you can simply abort without writing the output file. The information in the output file will be out of date, but at least it won't be lost.

- If you want to implement two-way syncing by having both Fetch and Dump conduits for a database, consider writing a single program that implements both flavors. Parts of the code will likely be the same, and you'll be more likely to keep the two mutually compatible.

  For instance, the `kab` format only has a single "telephone" field, and does not distinguish between home and work phone numbers. When converting a Palm database to a `kab` file, you can simply concatenate all of the various phone fields. When doing the reverse, however, you should look at each phone number in turn and see if it appears in *any* phone field in the Palm database.

  You're less likely to forget this if you only have one program.

- Fetch conduits should run quickly: when they run, the Palm is in the cradle and the user is waiting. It would be nifty to have a Fetch conduit that downloads the latest headlines from the web, but do you really want to wait for the conduit to surf to `http://www.washingtonpost.com` every time you sync?

  If you want to do this sort of thing, consider setting up a `cron` job that'll fetch the latest headlines every hour and save the results to a file. Then your Fetch conduit can quickly read this file and not keep the user waiting.

---

[1] If you uncheck the "Save archive copy on PC" box, the data in the record will be deleted to save space, but the Palm will still save the record header until the next sync.

Dump conduits, on the other hand, run after the main sync, after the Palm has displayed the "HotSync complete" message. The user can pick up the Palm and walk away, even if the Dump conduits are still running.

- The database that your conduit reads or writes may have other conduits associated with it. Try not to mess things up for them. The fundamental rule of syncing is, "Don't delete unless you're sure."

  For instance, if you have a conduit that updates the Address Book from a company-wide database, don't just delete every address that's not on the list: you'll delete private addresses as well. In this case, it would probably be best to consider only addresses in the "Business" category, and leave the other ones alone.

3 Specification

ColdSync allows the user to extend its functionality by means of *conduits*. A conduit is a program that is run from ColdSync, and which interacts with ColdSync according to a well-defined protocol.

Conduits come in multiple *flavors*. ColdSync runs conduits at several points during its execution. The flavor of a conduit indicates the intended purpose of the conduit. See Section 3.4 [Conduit Flavors], page 23.

ColdSync communicates with conduits by sending information to the conduit's standard input (see Section 3.1 [Conduit Input], page 20), and by reading results from the conduit's standard output (see Section 3.2 [Conduit Output], page 22). This information is mostly in human-readable text form.

A sync consists of the following phases (somewhat simplified):

1. Initialization

   ColdSync starts up, reads its configuration file(s), and performs basic initialization.
2. User presses HotSync button

   This indicates the beginning of the sync.
3. Gather information

   ColdSync gathers information from the Palm, such as the list of databases.
4. Fetch preferences

   ColdSync fetches from the Palm any preferences specified in the conduit configuration. Note that in the current implementation, ColdSync fetches all preferences listed in all conduits' configuration, even if those conduits will not be run.
5. Run Install conduits and install new databases (before sync)

   If ColdSync was told to install new databases before the main sync (the `-z` option was not specified), it now examines each database in the install directory and runs its Install conduits.

   Afterwards, if any databases remain in the install directory, ColdSync uploads them to the Palm. If the upload was successful, ColdSync deletes the file from the install directory, and adds an entry for the new database to its list of databases installed on the Palm.
6. Run Fetch conduits

   For each database on the Palm (including ones that have just been installed), ColdSync runs its Fetch conduits. Fetch conduits are expected to create or modify the databases in the backup directory. See Section 3.4.2 [Fetch Conduits], page 23.
7. Run Sync conduits (main sync)

   ColdSync runs Sync conduits for each database on the Palm. If no other conduit was specified for a given database, ColdSync defaults to running the generic conduit. As each database is synchronized, the message "Synchronizing *Database*" appears on the Palm. If no backup file exists, ColdSync creates one by downloading the database.

   ColdSync then checks all of the files in the backup directory: if a file does not correspond to any database on the Palm, that file is moved to the Attic directory.

8. Run Install conduits and install new databases (after sync)

   If ColdSync was invoked with the `-z` option, it will run Install conduits and install new databases after the main sync. See above.

9. Close the connection

   The message "HotSync complete" appears on the Palm. The user may remove it from the cradle.

10. Run Dump conduits

    For each database on the Palm (including the ones that were just installed), ColdSync runs its Dump conduits. Dump conduits are expected to read the databases in the backup directory. See Section 3.4.3 [Dump Conduits], page 24.

11. Clean up

    ColdSync cleans up and terminates.

If the configuration file calls for more than one conduit of the same flavor to be run for the same database, the conduits will be run in the order in which they are listed in the configuration file.

ColdSync makes no guarantees as to the order in which different databases' conduits will be run, nor is there any guarantee that all of the conduits for one database will be run before the first conduit of another database.

3.1 Conduit Input

ColdSync passes information to conduits in three ways: as command-line arguments, in a series of *headers* and *preferences* on standard input, and through a designated special file descriptor (see Section 3.3 [SPC], page 22).

3.1.1 Command-line Arguments

A conduit is invoked with two command-line arguments. The first is the string `conduit`. This argument will always be present when the conduit is run from ColdSync.

The second argument designates the conduit flavor: the string '`fetch`', for Fetch conduits, the string `dump`, for Dump conduits, the string '`sync`', for Sync conduits, or the string `install` for Install conduits. (see Section 3.4 [Conduit Flavors], page 23).

3.1.2 Standard Input

ColdSync writes a series of *headers* and *preference values* to the conduit's standard input. Each header is a line of the form

> *Field*: *Value*

Field is a string indicating a particular type of header. It may consist of upper- and lower-case letters, digits, hyphens, and underscores. The first character may only be a letter or an underscore. The length of *Field* may not exceed 32 characters. Fields are case-sensitive: `Foo` is not the same as `FOO`.

The field is followed by a colon, a space, the *Value* string, and a newline character. The entire line may not be more than 255 characters in length, including the field, but not including the terminating newline character.

Note that a single space follows the colon. Any other whitespace is part of the value. This allows a conduit to accept an all-whitespace header value (e.g., a paragraph indentation string).

Value is a string. It may not contain any newline characters. No other restrictions are placed upon its contents, although conduit writers are encouraged to use human-readable strings.

There is currently no provision for multi-line header lines, nor for encoding methods such as "Quoted-printable."

The end of the header is indicated by a blank line, i.e., two newlines in a row.

Since preference data is binary, preferences are passed to the conduit in two parts: first, a set of `Preference` headers specify the preference's creator, ID, and length.

Then, after the two newlines indicating the end of the headers, comes the preference data. This is simply the concatenation of the binary data for all of the preferences referred to by the `Preference` headers, in the order in which those headers appeared.

3.1.3 Predefined Headers

Certain header fields have predefined meanings and formats:

`Daemon`

: The `Daemon` header identifies the program that ran the conduit. ColdSync uses the string `coldsync` as the value.

: According to the terms of the Artistic License, a person may modify ColdSync. If the modified version is incompatible with the Standard Distribution of ColdSync, it should use some string other than `coldsync` to identify itself.

`Version`

: The `Version` header specifies the version of the program identified by `Daemon`.

`InputDB`

: The `InputDB` header specifies the full pathname to the database file that the conduit should use as its input. This header is mandatory for Sync and Dump conduits, and is optional for Fetch and Install conduits.

: The file specified by `InputDB` might not exist, or might not be readable by the conduit. In such cases, the conduit should fail gracefully.

`OutputDB`

: The `OutputDB` header specifies the full pathname to the database file that the conduit should use as its output. This header is mandatory for Sync, Fetch, and Install conduits, and is optional for Dump conduits.

: The file specified by `OutputDB` might not exist, in which case the conduit may create it. If it exists but is not writable by the conduit, the conduit should fail gracefully.

`Preference`

A `Preference` header identifies a preference being passed to the conduit. Each `Peference` header is of the form

`Preference:` *creator*`/`*ID*`/`*length*

where *creator* is the preference's four-character creator, *ID* is its numerical identifier (decimal), and *length* is the length of the preference data following the header (decimal).

`SPCPipe`

The `SPCPipe` header specifies the number (decimal) of an open file descriptor. This file descriptor may be used for SPC communications (see Section 3.3 [SPC], page 22).

Other headers may be passed to the conduit, so the conduit should be prepared to handle them.

If multiple `InputDB` or `OutputDB` headers are sent to a conduit, the last value sent is the authoritative one. It is an error to send more than one `Daemon` or `Version` header.

3.2 Conduit Output

A conduit may write lines to standard output. Each line should be of the form

NNN Message

where *NNN* is a three-digit status code (see Section 3.5 [Status Codes], page 24), and *Message* is a human-readable string. A single space separates the status code from the message.

Each line may be up to 255 characters in length, counting the status code, but not counting the terminating newline. There is currently no provision for multi-line messages.

The three-digit status code indicates some condition that the conduit wishes to report to ColdSync, such as successful or unsuccessful termination, a warning, or an informational progress report. The meanings of various status codes are defined in Section 3.5 [Status Codes], page 24.

ColdSync does not attempt to parse the message string. It is intended solely for human readers.

A conduit may print any number of lines. The status code of the last line printed by the conduit before it exits indicates the final exit status of the conduit, i.e., success or failure.

Only 2*yz*, 4*yz*, and 5*yz* status codes may be used for the exit status.

3.3 SPC

The Serialized Procedure Call (SPC) is a protocol that allows a Sync conduit to communicate with the Palm. A Sync conduit is passed an `SPCPipe` header that gives the number of a file descriptor over which it can communicate via SPC (see Section 3.1.3 [Predefined Headers], page 21).

SPC is similar in spirit to RPC (Remote Procedure Call): the conduit sends an SPC request over the SPC file descriptor, and receives a response.

SPC requests and responses have the same form:

```
unsigned short op;
unsigned short status;
unsigned long len;
// len bytes of data
```

where `op` is an opcode indicating which operation to perform; `status` indicates whether the operation was successful or not; `len` gives the length of the data following the header.

Supported values for `op` are given by the *SPCOP_** constants in '`spc.h`' in the ColdSync distribution. For known values for `status`, see the *SPCERR_** constants in that file.

All numeric values are given in network byte order. A `short` is 16 bits long; a `long` is 32 bits long.

`status` is ignored in a request packet.

In a response packet, `op` is the opcode of the request.

The following SPC operations are currently defined:

- `NOP` No-op. This does nothing, neither sends nor receives any data from the Palm, and is mainly useful as a connectivity test between the conduit and ColdSync. The response to a `NOP` has 0 bytes of data.
- `DBINFO` Get database information. Returns a `struct dlp_dbinfo`, similar to those returned by the `dlpReadDBList` request, except that the SPC request returns the copy that ColdSync has already fetched, so it does not involve extra communication with the Palm over a slow serial connection.
- `DLPC` This is the most powerful SPC request. The data for a `DLPC` request is a raw DLP request that will be sent to the Palm. The response is the raw response, as received from the Palm.

3.4 Conduit Flavors

There are currently four conduit flavors: Sync, Fetch, Dump, and Install.

3.4.1 Sync conduits

Sync conduits run during the main sync. The intended purpose of a Sync conduit is to ensure that a database on the Palm and its backup on the desktop both contain identical, up-to-date information. The generic conduit is a special instance of a Sync conduit.

Sync conduits have the unique property that they run while ColdSync is connected to the Palm, and so may communicate with it by using the SPC protocol (see Section 3.3 [SPC], page 22).

3.4.2 Fetch Conduits

Fetch conduits run before the main sync. The purpose of a Fetch conduit is to create or modify the backup copy of a database before ColdSync compares it to the copy on the Palm. The conduit may, for instance, add new records to be uploaded to the Palm.

3.4.3 Dump Conduits

Dump conduits run after the main sync. The purpose of a Dump conduit is to examine a database in the backup directory after it has been synchronized with the copy on the Palm. Usually, a Dump conduit will write the contents of the database in some other format, e.g., that of some other application.

In most cases, a Dump conduit will only need to read its database. If necessary, however, it may modify the database. For instance, a Dump conduit for the Palm Mail application may send out every message in the Outbox, then delete it from the database.

3.4.4 Install conduits

Install conduits run just before new databases are uploaded to the Palm, which may happen either before or after the main sync. The purpose of an Install conduit is to examine a database that is about to be installed, and possibly delete or modify it, or create a new one.

3.5 Status Codes

The status codes that a conduit sends to ColdSync should be of the form of a three-digit decimal integer xyz.

The first digit, x, indicates a general message class (success, failure, informational message, etc).

The second digit, y, represents a category within the message class (file error, system error, etc.)

The third digit, z, indicates a specific condition (out of memory, no such file, etc.)

The precise meaning of most status codes is still undefined. Only the following classes have been defined:

$0yz$ — Debugging messages. These will normally not be shown to the user. The precise circumstances under which they will be shown to the user have yet to be defined.

$1yz$ — Informational messages. These will normally be shown to the user, but do not indicate that anything is wrong.

In future versions of ColdSync, certain $1yz$ codes may acquire standard meanings, and their associated text strings will have a recommended format. This should allow real-time updates of the form "NN% complete" that can be displayed on the user's desktop.

$2yz$ — Successful completion.

There might conceivably be multiple forms of success. In the meantime, the recommended code for a plain, ordinary success is 202.

$3yz$ — Warning. The $3yz$ status codes indicate that something is wrong, but that the conduit has managed to recover.

301 — Unspecified warning.

Indicates that something odd has occurred, but does not specify what. This is the default warning printed by the `ColdSync.pm` module if no status code was provided.

4*yz* ColdSync (caller) error. A 4*yz* code indicates that the conduit has failed because it was given improper input, e.g., invalid or missing command-line arguments, nonexistent `InputDB` file, unsupported version of ColdSync, and so forth.

401 Unspecified Error.

402 Lost connection to Palm.

5*yz* Conduit error. A 5*yz* code indicates that the conduit should have been able to perform its function, but couldn't. Reasons include running out of memory, segmentation violation, inability to parse an apparently-correct `InputDB` file, etc.

501 Unspecified error.

Indicates that the conduit was not able to terminate for some reason other than improper input, but does not specify the cause. This is the default error code used by the `ColdSync.pm` Perl module.

6*yz*-9*yz* These error codes are currently undefined, but may be used in future versions. Until then, they should not be used.

The last status code sent by the conduit before exiting gives the exit status of the conduit as a whole. That is, if a conduit exits successfully, the last thing it prints should be a 2*yz* status code.

Only 2*yz*, 4*yz*, and 5*yz* status codes may be used for the final exit status.

Glossary

'`.coldsyncrc`'
: The per-user configuration file for ColdSync. Specifies the serial device to which the cradle is attached, and the list of conduits to run for each database.

Archive file
: A file containing records that have been deleted, but which were marked for archival, usually by checking the "Save archive copy on PC" box.

Archive directory
: The directory in which ColdSync stores archive files. `~/.palm/archive` by default.

Archived record
: A record that has been deleted on the Palm, but saved in an archive file on the workstation.

Attic directory
: When ColdSync finds a file in the backup directory that has no corresponding database on the Palm, it moves the file to the attic directory, in case the database was deleted by mistake. The attic directory is `~/.palm/backup/Attic` by default.

Backup
: A file in the user's home directory, containing a copy of a Palm database.

Backup directory
: The directory in which ColdSync keeps backups of the databases on the Palm. `~/.palm/backup` by default.

Conduit
: An external program, run at specific times by ColdSync, which provides a "hook" by which the user may extend ColdSync's functionality.

Database
: PalmOS's equivalent of files. Often used interchangeably with 'files', if the database in question has been downloaded to a file on a workstation.

Database creator
: A four-letter string that identifies the application that created a database. Each application has a unique creator string.

Database type
: Either 'record database' or 'resource database'.

Deleted record
: A record that has been marked for deletion. Deleted records are not removed from a database until a sync.

Dump conduit
: A conduit that runs after the main sync, usually to convert a PDB to some other format.

Export
: To convert a Palm database into another format, often one readable by another application.

Expunged record
: A deleted record that need not be archived, usually because the "Save archive copy on PC" box was left unchecked when the record was deleted.

  The data portion of an expunged record is immediately reclaimed. The header, however, is not deleted until a sync.

Fetch conduit
: A conduit that runs before the main sync, usually to convert a file to a PDB.

Flavor
: One of several types of conduit. Each flavor serves a different purpose.

Header
: A set of input values that ColdSync passes a conduit on standard input.

Header field
: The identifier on a header line, as opposed to the header value.

Import
: To convert a file, typically one generated by another application, to Palm database format.

Install directory
: A directory in which ColdSync looks for new databases to upload to the Palm. `~/.palm/install` by default.

Main sync
: The phase of a sync during which ColdSync compares the databases on the Palm with their backups in the backup directory.

Record database
: A database that contains data records, as opposed to a resource database. Almost all of the "useful information" on a Palm is stored in record databases. Record database files have a `.pdb` extension.

Record identifier
: An integer that uniquely identifies a record in a record database.

Resource database
: A database that contains resources, as opposed to a record database. Resource databases tend to have a static structure. Palm applications are resource databases. Resource databases have a `.prc` extension.

Resource identifier
: An integer that uniquely identifies a resource among all of the other resource of the same type in a resource database. The resource identifier is chosen by the application programmer, and remains constant.

Resource type
: A four-letter string that identifies the type of data contained in a resource, e.g. executable code, bitmap images, menu entries, etc.

Serialized Procedure Call (SPC)
: A protocol that allows a Sync conduit to send DLP requests to the Palm.

Status code
: A three-digit integer by which a conduit can keep ColdSync informed of its situation, e.g., whether a conduit has succeeded or failed.

Sync
: *vi.* To compare the databases on a Palm with those in a backup directory, and make whatever changes are necessary so that the two copies have identical contents.

: *n.* The process of syncing.

Index

Table of Contents