Pyrite: A Framework for Palmtop Data Interchange

Rob Tillotson
robt@debian.org

June 8, 1999

Abstract

The Palm Computing platform was designed with tight desktop integration in mind. Pyrite is an Open Source toolkit which allows desktop applications to interact with Palm handhelds and their data. It is implemented in the Python programming language, and is deliberately quite different than the standard Palm Desktop software. It is designed for maximum flexibility; Python's loose, dynamic object structure is exploited to allow applications to easily extend Pyrite's capabilities. In this paper, I will explain the most significant aspects of Pyrite's design, and describe some of the issues encountered during implementation.

1 Background

Before the 1996 introduction of the Pilot connected organizer, pocket-sized computing devices fell into two basic categories: organizers and handheld computers. The difference was one of extensibility. Organizers, like the Sharp Wizard and Casio B.O.S.S., were essentially single-purpose devices: they had built-in software to track schedules and phone numbers and expenses, and there was very little chance of doing more. Handheld computers (such as the HP LX, Psion 3, and Apple Newton) were much more flexible, with larger memory sizes, expansion slots, and most importantly the ability to run a wide variety of add-on software.

Missing from all of these devices, however, was the sort of tight desktop integration found in the Palm handhelds. Starting with the very first Pilot 1000, the Palm handhelds have been designed to work closely with a desktop computer, acting as a mobile "window" into the data on the desktop. The handheld itself is intended to be cheap and disposable, with modest hardware[1] and and lightweight operating system. The handheld communicates with the desktop through a serial port, and comes with a drop-in docking cradle.

Palm calls their products "connected organizers," to highlight the differences between them and previous handhelds. In order to make the "connected organizer" paradigm work, Palm Computing designed a new operating system with modest resource requirements and integrated desktop communication. Two aspects of the Palm OS design are particularly important to desktop software developers:

Everything is a database. The Palm OS treats the contents of the handheld as a pool of simple, flat databases, with variable length records. There is no support for files as byte streams; applications must always work with databases one record at a time.[2] Even executable programs are simply databases, containing code and user interface resources.

[1] Current models have a Motorola 68K series CPU running at 16 MHz, and 1–4 MB of RAM.

[2] Palm OS 3.0 has a "streaming" interface which is similar to the C standard I/O library, but it is implemented atop the database system.

Record-oriented synchronization. The Palm OS uses its own protocol stack[3] to communicate with the desktop. The highest-level protocol, DLP, provides a set of operations which closely correspond to the database interface that programs on the palmtop use. The database manager on the palmtop supports synchronization by marking modified records and temporarily archiving deleted records.

The desktop side of the synchronization system is part of the Palm Desktop, a software package provided with every Palm handheld. Third party software can extend the Palm Desktop's synchronization capabilities by adding *conduits*, which are program modules that can transfer information between a docked handheld and some other data source.

Because the Palm Desktop software is only available for Windows and Macintosh, Palm handheld users who do not use one of those platforms have had to develop their own replacements for it. Most of these are based on pilot-link[5]. Originally released in late 1996, pilot-link includes a portable library that implements the Palm synchronization protocol stack, and a set of core utilities which use that library. Many other applications have been built upon pilot-link; a few of the more substantial and/or ambitious ones are listed in the bibliography.[4] Pyrite uses pilot-link for communication with the palmtop, but does not use any of its other features.

2 Motivation

When I began working on Pyrite in late 1997, my original intention was to create a Palm Desktop-like application using Python as the implementation language and Tkinter as the graphical user interface. At the time, the only Python language interface to the Palm protocols was the one provided with pilot-link, a relatively straightforward translation of the pilot-link library interface to Python 1.4. As I began to enhance that interface for Python 1.5, I decided to use it as a vehicle to explore a design quite different from the Palm Desktop's conduit system.

To understand why Pyrite is not a copy of the standard synchronization interface, it may be helpful to think in terms of "parts" and "glue." The underlying design philosophy of Unix is one of tool-building, in which complex systems are built from smaller parts. Today, the parts we use are bigger and the interconnections between them are more complex; the "glue" in today's systems is often a dynamic, interpreted language like Python or Perl. Pyrite's job is to bring the information on the palmtop within reach of Python, making it available in a way that is both convenient and powerful.

The standard synchronization interface connects specific palmtop databases to specific desktop data sources, in the context of a HotSync session. The relationship is one-to-one, and the conduit interface simply provides a convenient way to move data back and forth. Pyrite does that as well, but decouples the act of communication from its content: synchronization is optional, applications can do everything conduits can do, and palmtop data can be exposed to Python without requiring that a corresponding desktop application exist.

From the viewpoint of the Palm application developer, this can be very significant. Consider what happens after a palmtop application is developed: if it can be used to collect a significant amount of data, a desktop interface to it will soon become important. Using the standard conduit interface, this means that the developer is faced with the prospect of writing both a complete desktop application and a conduit, using different development tools (and possibly a different language) than those used to make the palmtop application.[5] Not surprisingly, most third-party appli-

[3] See [8], [6], and [7].
[4] See [9] and [10].

[5] Palmtop applications are usually coded in C, cross-compiled using the GNU C compiler or with a special version

cations (especially those from small and/or non-commercial developers) don't come with conduits or desktop software, and their data is trapped inside the palmtop.

Pyrite places an abstraction layer between the palmtop and desktop applications. Information from a palmtop application can be made available in a form that is amenable to further manipulation in Python, by adding interface modules which pack and unpack records from the application's databases. This is considerably easier than developing a complete desktop application, and exposes the data on the palmtop to Python's full capabilities as a glue language. Other facilities can be built on top of this data abstraction layer, so that the implementation details of the palmtop application can be hidden even from its own conduits.

My goal in designing Pyrite, therefore, was to provide tools for palmtop data interchange in a powerful, flexible way. Making this possible requires that Pyrite focus on several important tasks:

- *Accessing databases in an intelligent manner,* using standard Python interfaces when possible and hiding implementation and storage details from the application programmer.
- *Organizing palmtop-related data in a logical fashion,* treating collections of databases the same no matter where they are (on the local disk or on the palmtop).
- *Supporting standard Palm OS data formats,* including the prc/pdb file format (used to store and transfer Palm OS databases on desktop systems) and the databases used by the built-in applications.
- *Allowing easy extension,* so that new palmtop applications can be supported and new conduits can be added with a minimum of unnecessary code.

of CodeWarrior; conduit development on Windows requires Microsoft Visual C++ or Java.

The rest of this paper will describe the elements of Pyrite's design and implementation which enable it to perform these tasks.

3 Design

3.1 Plug-Ins

Python's loose, dynamic nature was a major influence in Pyrite's overall design. The externally visible interface to Pyrite is a shallow but wide tree of classes, several of which serve as the base for an entire family of interchangeable components. Pyrite can be roughly divided into two parts: a component framework and a set of components for desktop/palmtop integration.[6]

Pyrite's components are called *plug-ins*. Plug-ins are essentially just Python modules and packages, but Pyrite builds upon the Python package system by providing more convenient access to plug-ins from within a running application. There are several important differences between plug-ins and ordinary modules and packages:

- *Plug-ins are organized by function, not by location.* A collection of plug-ins sharing a common interface can span many physical locations; an application does not need to know or care where a particular plug-in is installed in the Python package tree. New capabilities can be added to a particular Pyrite installation without changing Pyrite itself.
- *Plug-ins are instantiated automatically.* When an application loads a plug-in, the result is an object which serves as the plug-in's visible interface. When many plug-ins share the same visible interface, applications can use them interchangeably.

[6]Because the Pyrite component framework is of sufficient generality to be useful on its own, it is also available separately.[11]

- *Plug-ins are automatically configured.* User-visible options in a plug-in are set to the user's preferred values when the plug-in is loaded, and whenever the application's configuration changes. (For example, if there are different option values for particular palmtops, they will be set appropriately depending on which palmtop is in the cradle.)
- *Plug-ins can be discovered at runtime.* Applications need not incorporate any knowledge about which specific plug-ins are available, because the list of available plug-ins (and their properties) can be obtained at any time.

A plug-in is nothing more than a regular Python module, and a collection of plug-ins is just a standard Python package (or packages). When the plug-in loader finds a requested module, it looks for a class inside it which inherits from the Plugin class, and makes an instance of that class to serve as the application's interface to the plug-in. A reference to the interface object is kept in a cache for the life of the application, and reconfigured when necessary.

3.2 Databases

3.2.1 Blocks

The Palm OS takes a relatively simple approach to databases: it doesn't care what is in them, as long as their contents can be divided into records. On the handheld a record is just a memory block; it is up to the application to impose whatever structure is required. Each record has a unique ID, a category (a small integer from 0 to 15) and a few attribute bits, but otherwise the handheld's data manager makes no attempt to structure the contents.

As a result, it is impossible to algorithmically infer anything about the structure of a database from its contents — in order to make a particular database useful, it is necessary to unpack the records into a more meaningful structure when reading them from the database, and do the reverse when writing them.[7] The same is true of other record-like data blocks, detailed discussion of which is beyond the scope of this paper.[8]

Pyrite encapsulates this behavior in the Block class (and its subclasses, such as Record and Resource); a Block is similar to a Python dictionary, except that the possible key/value pairs are restricted (each represents one field in the data block) and can transform its values to and from a formatted string of bytes suitable for storage in a database.

3.2.2 Inner and Outer Layers

Pyrite's database interface is divided into two layers. An active database is represented by two object, one "wrapped" around the other. The outer object is the one applications actually interact with; its visible interface is much like that of a regular Python list, with additions to support the unique needs of the Palm database paradigm. Inside that, a back-end object manages access to the actual database; the back-end interface is less Python-like, and more like the native one.

This division is intended to make Pyrite more flexible and customizable. The inner object treats records in the database as opaque blocks of arbitrary bytes, and concerns itself only with how and where those blocks are stored. The outer object assigns structure and meaning to those arbitrary blocks of data, wrapping them in Block objects without regard to where or how they are actually stored. As a result, the code specific to a particular

[7] Here, the lightweight nature of the Palm platform is helpful: most Palm applications store their data simply, using standard C types and null-terminated strings, making it easy to understand and manipulate in Python.

[8] In brief, the other block types include resources (stored in special databases, each has a type and numeric ID), appblocks (each database has one, an extra record stored outside the normal sequence and normally used to hold database-wide information), and preferences (application-specific data stored in a system database).

Palm application (packing and unpacking its internal record formats) only has to be written once, and it will work with any and all implementations of the low-level database interface.

3.2.3 Implementation Issues

Unfortunately, implementing the back-end database interface was more difficult than I expected. Neither the DLP protocol nor the prc/pdb local file format are well-suited for general purpose, random access to the contents of a database — DLP is meant for synchronization, and the prc/pdb format is meant as a way of transferring whole databases at once. (This is why the standard Palm Desktop doesn't store its local data in prc/pdb files, but in a more flexible data structure provided by the Visual C++ runtime library. Similarly, Pyrite works with prc/pdb files using an in-memory cache, which bypasses the most annoying limitations of the format.)

As a result, Pyrite database objects exhibit differences in behavior, even though the interface to all of them is ostensibly the same. Some back-end databases automatically assign unique IDs, others do not. Some can insert new records at any position in the database, others can only append them to the end. It was impossible to hide these differences behind the outer interface; instead, each database has a set of properties which describes these small details. An application can check these properties and modify its behavior appropriately, or request a database with a certain set of properties, if one is available. Although this is an imperfect solution, it seems to work reasonably well in practice.

3.3 Stores

The Pyrite database layer is primarily concerned with the contents of individual databases. Management of database storage is handled by another interface layer, the *Store*. A Store is an abstract representation of a related collection of databases, such as the contents of the palmtop. Stores follow the Palm OS conventions for file system organization (a flat pool of databases referenced by name, creator, and type). The purpose of the Store object is to map this view onto an underlying representation, providing a consistent interface to applications no matter where the databases actually are.

Most Stores correspond to a particular kind of back-end database object. For example, each Store which works with prc/pdb files (there are several) uses the caching back-end described previously, and the DLP store uses a back-end object derived from pilot-link. To deal with differences in back-end capabilities, the application may provide "hints" to a Store about each database it opens, including requests for particular behaviors, suggested database header values, and other Store-specific information.

3.4 Palmtop Application Support

Above, I described how Pyrite uses Block objects to translate data between its stored form and an expanded, dictionary-like representation. Determining the proper kind of Block to use, however, is another matter entirely, and that is where application support modules enter the picture.

Fully supporting a particular Palm application requires several bits of knowledge about that application, for example:

- The name, creator ID, type, and other identifying information of its databases
- The format of records, appblocks, and/or resources in its databases
- How its databases are organized, if different than the usual conventions
- What preferences it uses, and their format

For each palmtop application Pyrite supports, all of this knowledge is encapsulated in an application support module. A typical application support module contains one or more Block subclasses — for example, the Mail application module has four: one representing a record in the mail database, and the other three representing preferences — and at least one custom outer-layer database class which uses them. Each application module is also a plug-in, with an interface that allows the opening and creation of databases, reading and writing of preferences, and so forth.

Early versions of Pyrite used an automatic classification scheme to detect supported databases. After importing App.Address, for instance, Pyrite would be made aware of Address databases; then, when the application opened 'AddressDB', the returned database object would be treated as an address database. In more recent releases, I removed that system in favor of the current one, which requires the desktop application to explicitly load the desired plug-in and request that it open the database. While the current scheme is not as convenient as the previous one, it places the responsibility for database classification where it belongs: inside the one module that contains all of the other application-specific knowledge.

3.5 Synchronization

Like the Palm Desktop, Pyrite's synchronization process involves the running of conduits. Conduits are plug-ins, loaded in the same manner as other parts of Pyrite.

During synchronization, the conduits in use are called in turn, in three passes. The first, pre-sync pass allows the conduits to do advance preparation, such as gathering data from an outside source. Actual synchronization is done during the second pass; the serial connection is opened on demand, the first time it is used, and closed at the end of the pass. The third and final pass allows the conduits to do post-processing and cleanup. The reason for this three-pass process is efficiency of communication: using the serial connection is "expensive", in both battery power and the user's time, and it should be unnecessarily left idle. Since the connection to the handheld is open only during the second pass, the first and last passes can be used to do time-consuming work without consideration of connection time.

There is, however, one small inconsistency in this type of conduit structure. Until the connection is opened, it is impossible to tell which palmtop is in the cradle. It is possible to guess the identity of the palmtop in advance, based on either a user-supplied default or on the identity of the most recently seen palmtop, but until the synchronization pass begins there is no way to accurately know which palmtop will be present. As a result, conduits have to be careful not to do anything in the pre-sync pass that might be affected by a change of identity during the other two passes. Dealing with this inconsistency is still an open issue in Pyrite's design; one possible solution might be to do the pre-sync pass outside of the normal synchronization sequence, once for each known palmtop (an ideal application for tools such as cron or at).

Pyrite does not impose any particular internal structure on its conduits, but it does provide help for basic synchronization tasks. Like the Palm Desktop, Pyrite includes a generic synchronization method; given a palmtop database and two desktop databases (primary and archive), it synchronizes their contents according to the logic described in the official documentation.[2] In theory, it should be possible for the generic synchronization code to reconcile records at the field level, as long as Pyrite has full support for the corresponding palmtop application; at the time of this writing, however, the code to do that is not yet complete.

3.6 Pyrite Applications

Pyrite's plug-in services, along with others such as command-line parsing and configuration file support, are provided through an Application class. Each Pyrite-enabled application is wrapped in a subclass of Application. The application's core code is a method on this subclass, and Pyrite services are available through calls to other methods on the application object itself.

Much of the Application class is concerned with configuration. Like most complex programs, Pyrite has many settings which can be configured by the user or system administrator. In addition to system-wide settings (such as the name of a port associated with a docking cradle, or the location of a shared data directory), many applications and plug-ins have local settings. When an application starts up, its Application object loads option values from two configuration files, one in /etc and one in the user's home directory, storing them in a layered registry so that the user's settings override those set by the administrator. When a plug-in is loaded, it is configured based on the stored option values. The Application object also parses the command line, extracting settings provided as arguments.

Making matters more difficult, however, is the fact that palmtop identity makes a difference. Most non-trivial Pyrite applications require some access to information associated with a particular palmtop, so the Application object keeps track of the "current user". Making a connection to a palmtop can cause the "current user" to change, since there is no guarantee that a particular palmtop will be in the cradle. The Application class deals with this by reconfiguring itself and all plug-ins every time the "current user" changes, in case different settings are associated with each palmtop.

The other significant purpose of the Application class is to act as a context for plug-ins. When a plug-in is loaded from within an Application object, many of the plug-in's methods will call their counterparts in the Application object in preference to their own default behavior. This way, application-specific behavior is possible without requiring changes to existing plug-ins.

4 Future Directions

Pyrite began as a simple refinement to pilot-link, and grew far beyond my initial expectations. While it is (in some sense) relatively complete, there is still much work to do. In addition to the ever-present need for more and better documentation, there are several potential improvements which appear to be worth pursuing in the long term.

First and foremost, Pyrite needs to support as many palmtop applications as possible. It is not necessary to connect the palmtop application to any particular desktop software; simply exposing its data to Python will make future integration easier. The work involved in adding an application support module is not difficult; instead, the primary obstacle to growth in this area is the closed nature of many palmtop applications. Open access to data on the palmtop is vital not only to the future of Pyrite but to the future of Palm/Unix compatibility in general, and gentle evangelism of palmtop developers about the benefits of open, documented data formats may prove to be very beneficial.

Next, Pyrite could be made less dependent on Python. Tying a library to a particular language can be a disadvantage even in the relatively flexible world of Open Source, so one of my long-term goals for Pyrite is to make it available to applications written in other languages. In the short term, the Pyrite class structure could be translated directly into another language, using the existing Python code as a prototype. Unfortunately, many popular languages are not amenable to a direct translation, and dynamic loading of classes is tricky in most compiled languages. In the long term, the most promising possibility is to use a distributed object system (CORBA, for example) to extend Pyrite's reach, and to allow Pyrite components to be used by programs in languages other than Python.

Finally, I plan to add more user-level support to Pyrite, in the form of graphical interfaces to most of Pyrite's functions. Other user interface improvements, including easier configuration and installation, will also be important. Pyrite could become the basis for a complete desktop application suite, using plug-ins to enable it to work with data from many palmtop applications. While it is not my goal to make Pyrite the only interface to Palm connected organizers and their data, I aim for it to be the most powerful and flexible one.

Availability

Pyrite and related tools are free software; most is under the GNU Library General Public License, but parts are also available separately under a more liberal license like that of the Python interpreter and libraries. The source code can be downloaded from the Pyrite home page:

http://purl.oclc.org/net/n9mtb/cq/

Two Pyrite-oriented mailing lists are available: pyrite-announce is moderated and carries announcements only, and pyrite-discuss is open to public discussion. To subscribe, send a message to pyrite-announce-subscribe@egroups.com or pyrite-discuss-subscribe@egroups.com, respectively.

References

[1] 3Com Corporation. *Conduit Developer Reference*.

[2] 3Com Corporation. *Palm Conduit Programmer's Guide*.

[3] 3Com Corporation. *Palm OS Programmer's Companion*.

[4] 3Com Corporation. *Palm OS SDK Reference*.

[5] Kenneth Albanowski et.al. pilot-link. ftp://ryeham.ee.ryerson.ca/pub/PalmOS/.

[6] Kevin L. Flynn. Unix and Pilots part III: The minimal HotSync. *Handheld Systems*, 5(3), May/June 1997.

[7] Kevin L. Flynn. Unix and Pilots part IV: The PADP layer. *Handheld Systems*, 5(4), July/August 1997.

[8] Kevin L. Flynn. Unix and Pilots: The protocol stack, part 1. *Handheld Systems*, 5(2), March/April 1997.

[9] Bharat Mediratta. PilotManager. http://www.moshpit.org/pilotmgr/.

[10] Dan Pilone. KPilot. http://www.slac.com/pilone/kpilot_home/.

[11] Rob Tillotson. Sulfur. http://purl.oclc.org/net/n9mtb/cq/sulfur/.

[12] Rob Tillotson. Palmpython: Unix conduit tools. *Handheld Systems*, 6(6):22–26, November/December 1998.