

FastPak for Java

Advanced Users Guide

Legal Information

All Software and Computer Systems Company, LLC logos shown in this document are a trademark (TM) of Software and Computer Systems Company, LLC. *Java JWaveScope* and *FastPak for Java* are trademark (TM) of Software and Computer Systems Company, LLC. All software produced and licensed by Software and Computer Systems Company, LLC is copyright© Software and Computer Systems Company, LLC **2005 - 2007**. The contents of all pages and images contained in this web site are copyright© Software and Computer Systems Company, LLC, **2007**,

Sun, Sun Microsystems, Solaris and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. X Windows and the X Window System are trademarks of the X Consortium. UNIX is a registered trademark in the United States and other countries of the X/Open Company, Ltd. Apple Macintosh and OS X are trademarks of Apple Computer, Inc. Novell is a registered trademark of Novell, Inc., and SUSE is a trademark of SUSE LINUX Products GmbH, a Novell business. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Motif is a registered trademark of The Open Group.

Unless explicitly stated, original products and services offered, sold, or licensed by Software and Computer Systems, LLC to customers are the exclusive right of Software and Computer Systems Company, LLC. Clients, users, or interested parties should not assume an affiliation exists between Software and Computer Systems Company, LLC and any of the computer manufacturers, operating system distributors, or other vendors that may be used in the production or completion of a work produced by Software and Computer Systems Company, LLC for a customer or product.

Table of Contents

| | |
|---|----|
| Introduction | 4 |
| Definition of Terms | 5 |
| The FastPak Directory Structure | 6 |
| Configuration Files | 9 |
| Examples of Application Configuration Properties Files In Use | 17 |
| Using the Console and Daemon Modes | 25 |

Introduction

Welcome to the *FastPak for Java* Advanced Users Guide!

This manual is intended primarily for advanced **FastPak** users, system administrators, and developers. This manual will provide a definition of terms used in this manual, describe the directory structure of a **FastPak** installation, describe how configuration files are used and laid out in a **FastPak** directory structure, provide some examples of how *application configuration properties files* are used, and describe how to use **FastPak's** console and daemon modes. **FastPak**, when operated using its graphical users interface, has its own, embedded help text which is extremely detailed, however most of the topics addressed in this manual will not be found in those help sections. Additionally, FastPak comes with a tool called the TransferUtility which makes the reconfiguration of application configuration properties files easy. The TransferUtility has its manual embedded into its help section, so it will not be addressed at all in this document.

Definition of Terms

The following list identifies terms used throughout this document and what they mean:

- **Jar application:** A “jar application” is a Java application that has been compiled into a series of class files and the class files are then archived into a single bundle with the Java archiving tool, “jar.” Users can often launch such an application from an OS GUI or a command line without needing to supply any external links, references, library paths , etc. for the application to run. This document will not go into any of the specifics of a jar based application or how to use the “jar” tool. A jar application is does not need to be tied to the hosting operating system since all references made in a jar application can be with respect to the jar archive itself. Jar files may also reference other jar archives as libraries.
- **Flat application:** A flat application is a group of class files that have been created using a Java compiler. These files are not archived, but rather exist in a directory structure, most often as a set of many individual class files. All references to the class files are absolute and require explicit references to the directories containing the class files if attempts are made to execute the flat application outside it's own directory structure.
- **Application Profile:** An application profile, or more briefly, profile, is the short name used by **FastPak** to identify a configured application. The application profile is associated with an *application configuration properties file*, which will have the same name as the profile, but will have a “.prop” extension. The name of the actual file launched by **FastPak** may be completely different from the profile name and its corresponding *application configuration properties file*. For example, the application profile called *LogoLib* has an *application configuration properties file* called *LogoLib.prop*, and yet the application that is accessed when *LogoLib* is executed is the jar file *SCSCLogoLibTest.jar*. This will be detailed in some of the following sections.

All illustrations provided in this manual were derived from a system using Apple Computer's OS X operating system. **FastPak for Java** is available for numerous operating systems and the appearance of any illustrations shown in this manual should be similar in content to those found on other operating systems and differ only in the appearance of such things as window decorations and the icons used to display such things as subdirectories.

The *FastPak* Directory Structure

The following image illustrates the layout of the directory structure used by ***FastPak***:

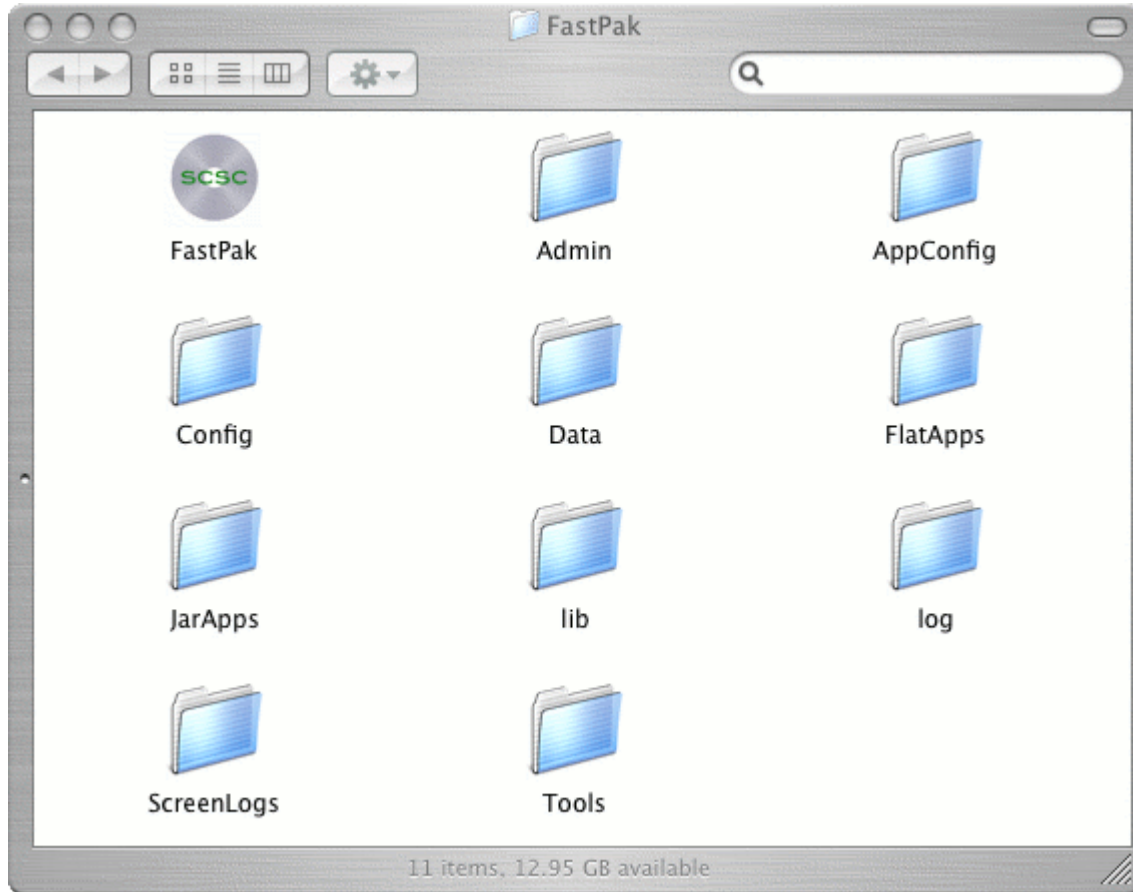


Figure 1. The *FastPak* directory structure.

The image above shows the ***FastPak*** application itself in the upper left corner of the diagram. The application itself may be, depending on the operating system, a directory itself (as in OS X's directories with a .app suffix), a link to another program, a configuration file, or script. The rest of the folders may be described as follows:

Admin – This is a folder where the installer is located. In future releases it may hold other tools as well.

AppConfig – This is the directory that holds all the application configuration properties files. ***FastPak*** reads these on start up to load all the profiles for the applications it has available. ***FastPak*** will also store all new application configuration properties files here. ***FastPak*** will not look in other directories for these files. This directory must be present for ***FastPak*** to work properly. Actual application binaries are not stored in this directory.

Config – This is where **FastPak** stores its own properties files and class path settings.

Data – This is a convenience directory where applications running under **FastPak** can store data if they so desire. After an initial installation, this directory will store a set of pictures used by the application profiles called “CarPics” and “SolarPics.” It is not mandatory that applications use this directory but it will make it easier to bundle items under **FastPak**.

NOTE: All applications executing under **FastPak** see the **FastPak** installation directory as their base directory, regardless of the actual location of the stored applications binary! Suppose a user has a hypothetical jar application called “JavaApp.jar” in a directory called “/Users/smith/MyJavaApp,” and suppose underneath this directory there is a directory for storing data called “MyData” (the full path would be “/Users/smith/MyJavaApp/MyData”). If the application uses no configuration information to tell it where to find the “MyData” directory because it was assumed by the developers that the “JavaApp.jar” application would always be launched from the directory where it's stored (“/Users/smith/MyJavaApp”) and the directory “MyData” was a relative reference to that directory, then when the application is configured to run under **FastPak**, it will never see the “MyData” and presumably the application won't work properly. It is recommended that in such cases the application be modified to use command line parameters (which can be configured and changed as needed under **FastPak**), use a properties file to tell the application where to find its data files, copy the data files to the **FastPak** directory, or on some operating systems, create a link or alias from the data directory to the **FastPak** base directory (the latter may not work in all cases depending on the operating system).

FlatApps – This is where flat applications can be stored. This directory is included in the **FastPak** initial startup class path assignments, thus any flat applications put there will not need to tell **FastPak** anything else about it, unless the application itself requires them. If a flat application resides outside this directory, then **FastPak** must have that class path added to **FastPak's** class path settings. For example, if a user named “smith” had a home directory of /Users/smith and installed a simple “HelloWorld” Java program in this home directory, then to configure the application, the user must perform a regular configuration of the application and add the class path “/Users/smith” to the **FastPak** class path settings. If the application had been put in this directory this step would not be needed. Any time **FastPak** has its class path setting modified, it must be restarted before they are recognized.

JarApps – This is where jar applications and libraries can be stored, and like the FlatApps directory just mentioned, is included in the **FastPak** initial class path settings. If a jar bundle is complete, and the manifest properly identifies all its class path settings, libraries, data, etc., then the types of restrictions that exist for flat applications (described above) will not apply. For example, if you took the “HelloWorld” program mentioned above in the FlatApps section, and made it into a fully executable jar file, then when the jar version of the application is configured, it would not require modifying any class path settings for **FastPak**. This directory is

particularly useful if you wish to make use of libraries in a jar format that will be referenced by numerous jarred applications.

lib – This directory is where **FastPak** stores libraries and scripts. The libraries include the primary jar library used by **FastPak** as well as that used by the **FastPak** API. Additionally, there is an application called *ConsoleConfigurator.jar* that will configure a script called *FastPakConsole* which is used to launch **FastPak** manually. This script is intended to be used for **FastPak** when it's been configured to use console mode, but it can also be used to launch **FastPak** in GUI mode. Console mode must be launched using this script and it must be run in a terminal or command shell.

log – This is where **FastPak** stores log files that are recorded when logging is turned on.

ScreenLogs – This is where **FastPak** will store log files that were selected and recorded from **FastPak** when using GUI mode. This directory is not used by console or daemon modes.

Tools – This is where **FastPak** stores tools. Currently these are the GUI version of the transfer utility (*TransferUtility.jar*) and the command line version of the transfer utility (*TransferUtilityCL.jar*).

NOTE: It is critical that the **FastPak** directory structure just detailed be maintained since **FastPak** uses it to determine and assign internal class paths and read properties associated with all applications to be launched using **FastPak**. A user can add all the directories to the **FastPak** base directory they desire, but none of those identified above should be deleted. Additionally, applications and scripts should not be moved out of this structure. If a user wishes to put the **FastPak** application on their desktops, they should do it via links, aliases, etc.

Configuration Files

There are two types of configuration files used by **FastPak**: those used by **FastPak** itself, primarily during startup, and application configuration properties files that tell **FastPak** the information it needs to execute an application. All of the configuration files for **FastPak** itself are stored in the directory called “Config” as shown in figure 1 above. All application profiles are stored in the “AppConfig” directory shown in figure 1 above. All files are simple Java properties files and they are **not** in XML format. This section will focus on the contents of these directories as well as the “Data” directory and their use.

If **FastPak** has been installed via the installer on a system, the directory called “Config” should contain the two files:

1. *FastPak.prop*
2. *FastPakClassPath.prop*

The file *FastPak.prop* contains the information needed by **FastPak** at start up. Here is an example of the contents of this file:

```
# FastPak configuration properties file
#Base directory for FastPak

BASE_DIR = /Users/smith/FastPak/Config/

# PORT_ID is the port that the FastPakController will use to communicate with
# remote machines. Default value is 20001. The number should be a base 10
# integer, and not in hex, octal, binary, or decimal formats.

PORT_ID = 20001

# CONTROLLER_STATE tells FastPak whether or not to bring the controller thread
# on line at start up. The only values that can go here are CONTROLLER_STARTING
# or CONTROLLER_DOWN. Default value is to run the controller with FastPak
# set at CONTROLLER_STARTING

CONTROLLER_STATE = CONTROLLER_DOWN

# DEFAULT_USER_INTERFACE_MODE determines which user interface to use. The
# options are CONSOLE_MODE, GUI_MODE, and DAEMON_MODE with GUI_MODE being
# default.

DEFAULT_USER_INTERFACE_MODE = GUI_MODE

# PASSWORD_PROTECTION is a boolean that tells FastPak whether or not to use
# passwords when accessing FastPak remotely via the ControllerThread. The
# default is true.

PASSWORD_PROTECTION = false
```

Figure 2. A *FastPak* Configuration File Example.

Each of the properties identified in the *FastPak.prop* file are described as follows:

BASE_DIR - This is the base directory for all configuration files for this specific installation example. If multiple instances of **FastPak** are installed for a user and different applications are bundled under each instance, then each installation must be in a unique directory. For example, in the case above, if the user, assumed to be “smith” from the directory path, wanted to install a second instance of **FastPak**, he could install another **FastPak** instance under a directory called “/Users/smith/fp2.” When the installer runs in this new directory, the **FastPak** instance will then have a base configuration directory of “/Users/smith/fp2/FastPak/Config” and both instances will be completely isolated from one another, however please note that if the Controller Thread is active on both, then each will need different port numbers as well.

PORT_ID - This is the network port number used by the *Controller Thread* for network access. The value 20001 is the default value **FastPak** selects when it's installed. It may be any valid value, but please note the following:

- If you intend to bundle separate applications and use them under different running instances of **FastPak** simultaneously, you must make sure each instance has it's own, unique port number. Since each instance of **FastPak** will look in it's own base directory, this means that if multiple instances of **FastPak** must be stored in different directories.
- Generally an end user will need to ensure that the port being used is not blocked by a firewall or in use by another application.
- Make sure the ports selected adhere to security policies when installed, and verify that firewalls will allow the use of an assigned port.

CONTROLLER_STATE - This tells **FastPak** whether or not to start the *Controller Thread* on startup. The options are:

- **CONTROLLER_STARTING** - This tells **FastPak** to launch the *Controller Thread* when it starts up
- **CONTROLLER_DOWN** - This tells **FastPak** not to launch the *Controller Thread* on startup

The default is **CONTROLLER_DOWN** but it can be brought up or down on demand via the GUI or the console mode. The *Controller Thread* is always active if **FastPak** is operating in daemon mode, regardless of the setting in this configuration file.

DEFAULT_USER_INTERFACE_MODE - This tells **FastPak** which user interface to use when it starts up. The values for this setting are **GUI_MODE**, **CONSOLE_MODE**, and **DAEMON_MODE** which tell **FastPak** to launch in GUI, console, or daemon mode respectively. The default after installation is GUI mode.

PASSWORD_PROTECTION - ***This field is not enabled for this release of FastPak, but it must remain present.*** It may be used in future releases of **FastPak**.

The *FastPak.prop* file is generated by the **FastPak** installer during installation and about the only time it will need manual editing is when the user has switched from GUI to either console or daemon modes and needs to switch back to GUI mode. In this case the values associated with DEFAULT_USERINTERFACE will need to be set back to GUI_MODE. **FastPak** currently provides no provisions for modifying any configuration files via the console or daemon modes, thus the files will need to be manually edited.

The file *FastPakClassPath.prop* is used to define base class paths and class path information for FastPak as shown below:

```
# FastPak class path settings properties

# This file is generated by FastPak, DO NOT EDIT THIS UNLESS YOU FULLY
# UNDERSTAND WHAT IT DOES.

#The following set of class paths are FastPak specific, THESE MUST NOT BE
#DELETED!!!!!!!

FAST_PAK_HOME = /Users/smith/FastPak/
FAST_PAK_JAR_APPS = /Users/smith/FastPak//JarApps/
FAST_PAK_FLAT_APPS = /Users/smith/FastPak//FlatApps/
FAST_PAK_LIBS = /Users/smith/FastPak//lib/FastPakLib.jar

#The following set of class paths are user specified, Please note that all of
#these class paths will be shared among all applications running under FastPak,
#BEWARE OF REDUNDANT NAMING OF CLASSES, It is STRONGLY recommended that
#developers review the development guidelines before adding class path entries
#that may use objects of the same name with different intent.

USER000 = /Users/smith/GoodByeWorld
USER001 = /Users/smith/HelloWorld
```

Figure 3. A *FastPak* Class Path Settings File Example.

In the figure above, the file contains the necessary **FastPak** class path setting as well as two add on class path settings. This file differs slightly from a file with a fresh installation in that additional class paths (USER000 and USER001) have been added. This will be explained below. The name/value pairs associated with this properties file are described as follows:

FAST_PAK_HOME - This defines base directory for **FastPak**.

FAST_PAK_JAR_APPS - This defines the preferred location where applications and libraries in “jar” format are stored.

FAST_PAK_FLAT_APPS - This defines the preferred location where flat applications are stored.

FAST_PAK_LIBS - This defines the location of the **FastPak** library, “*FastPakLib.jar*” as well as the API library and some launching scripts.

All of the settings just mentioned are included and generated by the **FastPak Installer** during installation. These should generally not be edited unless the user is absolutely certain they understand what they are doing. The other entries, “USER000” and “USER001” are described below, and they are not part of the **FastPak** initial configuration, but rather what will appear after a user has modified the **FastPak**

class path settings via the GUI.

USER000 and USER001 - These are class path settings that **FastPak** will add **Globally** to all applications running under **FastPak** after a user has added them. After an initial installation, these values will not be present. This information is provided to illustrate what will occur if a user adds two additional class paths after installing **FastPak**. This is most commonly done for flat applications. The identities USER000 and USER001 are assigned by **FastPak** when the user edits the class path setting via the **FastPak** GUI. The identity can be broken down into two strings, with the “USER” prefix identifying it as a custom class path for **FastPak** to add, and the suffix being a numeric string from “000” to “999.” **FastPak** will sequence the last three numeric contents of the suffix to account for up to 1000 class path entries (000 through 999). Note that the number of class path entries may be limited to well below this number based on the JVM and the operating system. The suffix strings are always sequential, meaning they will always be 000, 001, 002, 003 etc. If one is deleted, then the sequence is adjusted so there are no empty sequence values. For example, if a user initially added 5 class paths which produced names of USER000, USER001, USER002, USER003, and USER004, and later deleted those entries associated with USER001 and USER004, a review of the properties file after this change will be USER000, USER001, and USER002, and not USER000, USER002, and USER003. The prefix string “USER” must always be present and in uppercase, and the suffix string must always be numerical entries and in a continuous sequence. The preferred method of packaging applications for use with **FastPak** is to bundle them up as jar files with the manifests for the jar file defining the location of additional class paths.

With the **FastPak** configuration files now described, it's recommended the reader open up and look at the contents of the directories called “AppConfig” and “Data”. The “AppConfig” directory is where all application profiles are stored, and the “Data” directory will be where all data that applications running under **FastPak** can be stored. Use of the “Data” directory is not mandatory, and it's intended for storage of local data, not items such as databases. The following sections will describe the contents of these directories in detail.

The “AppConfig” directory contains *application configuration properties files*. If your directory is untouched after a **FastPak** initial installation, the contents of this directory should be as follows:

```
ActiveLogo.prop
Analyzer.prop
CarPics.prop
FourierAnalysis.prop
LogoLib.prop
RemoteAnalyzer.prop
RemoteFourierAnalysis.prop
RemoteSCSCLogo.prop
SCSCLogo.prop
SolarPics.prop
```

Figure 4. Directory Contents of the AppConfig Directory After a *FastPak* Installation

Looking at figure 4 above, a list of *application configuration properties files* is shown. **FastPak** does not directly access an application by its actual name but rather by its *application configuration properties file*. When a user wishes to execute an application under **FastPak**, the names of available applications presented to the user will not be the application names themselves but rather the names of the profiles (shown in figure 4 above) minus the “.prop” extension. For example, if you start up a newly installed and unmodified version of **FastPak**, select the “Control” menu item, followed by the “Run Configured App” pull down item, the following window should appear:

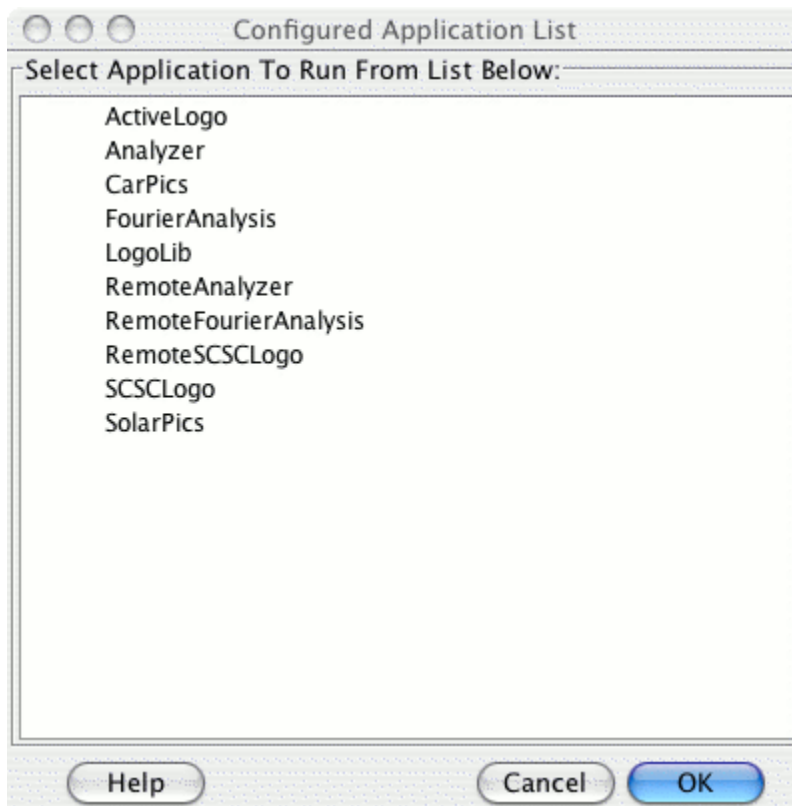


Figure 5. List of Available Application Profiles After a *FastPak* Installation

What should be obvious is that the list shown in figure 5 above matches that of figure 4 with the exception of the “.prop” extension. **FastPak's** designer chose to reference applications by a profile name instead of an actual application name for one simple reason: the entire purpose of **FastPak** is to run multiple applications under a single JVM, and this includes, provided **FastPak** design guidelines are followed, applications of the same name but with differing command line parameters. As will be shown shortly, the application profiles “CarPics” and “SolarPics” are the same application, but “CarPics” looks in the **FastPak** “Data”

directories for a set of pictures of cars, whereas the “SolarPics” is configured to look in the “Data” directory for pictures related in one way or another to solar material (sunsets, space pictures, etc.) Additionally all applications that start with the word “Remote” are simply copies of their corresponding local applications, but they are hosted on a remote server (the use of the word “Remote” is not mandatory, it could be any word as long as all profile names are unique). For example, the profile for *RemoteSCSCLogo* executes the same program as *SCSCLogo*, but the remote version is hosted on **SCSC's** web site.

application configuration properties files are usually configured using **FastPak's** GUI (this is done by selecting the menu item “Edit” followed by the “Configure New” pull down item). When the configuration is complete, **FastPak** generates the corresponding properties file. An example of such a properties file is show below:

```
# APP_NAME is a string - This is the actual name of the application .....
APP_NAME = logo.ActiveLogo

# APP_VERSION is a string - This is a user supplied version number .....
APP_VERSION = 1.0

# APP_PATH is a string - The base path of the jar or class file that FastPak
# will be executing.
APP_PATH = /Users/smith/FastPak/FlatApps/

# IS_JAR is a boolean, and will be true for jar file, false for full app
# SCSC recommends using jar files for all applications.
IS_JAR = false

# IS_REMOTE is a boolean - true = jar file from web server, false = local file
IS_REMOTE = false

# REMOTE_PATH is a server path for remote applications...leave blank for
# local apps
REMOTE_PATH =

# Hostname is the string name of the remote host, "localhost" for local files
HOSTNAME = localhost

# PRE_LOAD is a boolean - true if the app is to be pre-loaded, false if loaded
# on demand. CURRENTLY ALL APPLICATIONS ARE PRELOADED
PRE_LOAD = true

# GO_FLAG is a boolean - true if the app is to automatically execute .....
GO_FLAG = true

# INPUT_ARGS is a list of Strings - command line parameters for .....
INPUT_ARGS =

# APP_PRIORITY value sets the priority of the launch thread. Values .....
APP_PRIORITY = 5
```

Figure 6 A FastPak application configuration properties file Example.

In the example above, some of the comments that **FastPak** embeds in the file have been eliminated for brevity by replacing them with ellipses (.....). The following will provide details about each entry in the *application configuration properties file*:

APP_NAME - This is the real name of the application. In other words, if you were to

launch this program via a command line, this is entry is the name of the file you would use. In this case the application in question is a flat application, meaning it is not a jar file but rather a distribution of class files. The flag "IS_JAR" determines this as described below. If the application is a flat application, then the name will be that of the class containing the main() method, but if it's for a jar file it will be for the jar file name, and that name may be completely different from the class containing the main() method.

APP_VERSION - This is not a mandatory field, and it was put in place as a convenience in the event that one wishes to keep track of which application version is installed.

APP_PATH - This is the path to the application itself. If the IS_REMOTE flag below is set "true" then this entry is ignored.

IS_JAR: This flag tells *FastPak* whether or not the application is in a jar format or a flat application of distributed class files in a directory. If this boolean string value is true, then *FastPak* sees the application as a jar file, if it's false it's seen as a flat application.

IS_REMOTE - This flag tells *FastPak* whether the target application is hosted on another machine and is being accessed via URL or if it's just a local file. If this flag is true then any input for APP_PATH is ignored and instead REMOTE_PATH is used.

WARNING: *FastPak* is not a secure product. Hosting of applications for remote access by *FastPak* should be done where users can be assured of a secure network and connection. Please see the document titled Security Warnings Regarding *FastPak* for Java.

REMOTE_PATH - This variable will provide the path to the application on a remote web server. This directory will typically be with respect to the base web url for that server.

HOSTNAME - This value will depend on whether or not the application is remote or local. All local applications will be assigned the name "localhost" but all remote applications will need to put the name of the actual server into this field.

PRE_LOAD - This is a legacy value and is always set to true.

GO_FLAG - This value will tell *FastPak* whether or not to auto-launch the application when *FastPak* starts. If this value is set to true, then the application will automatically launch when *FastPak* starts. If it's set to false, then the application must be started when the user decides to start it .

INPUT_ARGS: These are command line arguments passed to the application at start up.

APP_PRIORITY: This is the application priority with respect to the *FastPak* kernel. The *FastPak* kernel runs at a priority of 10 (highest). The default value filled in by *FastPak* is 5, or normal. When using the *FastPak* GUI to configure applications, the value of "high" priority is not set to 10 but rather 8. This is to prevent an application from running at the same JVM priority level as the *FastPak* kernel. The only way to override this is to edit the application configuration properties files for the

application in question and manually change it to 10. *DO SO AT YOUR OWN RISK!!*

Examples of Application Configuration Properties Files In Use

The best way to get a good understanding of the *application configuration properties files* is to examine those that come as examples with the **FastPak** kit and are available once installed. Here are some examples that may be of interest to those wishing to have a more comprehensive understanding of **FastPak's** application configuration properties files:

Example 1: Compare the file CarPics.prop to SolarPics.prop - Under **FastPak**, one of these will show up on the application selection list (under the “Control” menu item) as “CarPics” and the other as “SolarPics.” Both are the exact same application, but the important difference (the difference that makes them function differently) is that they both accept different settings for INPUT_ARGS. The application associated with both of these is called “ImageViewer.jar” and it requires a name/value pair that will identify where the application can go to find a directory of images to cycle through.

If you look at SolarPics.prop, you will see something similar to the following (the format is operating system specific, but this would be typical of Linux or OS X installations):

```
imagePath=/Users/smith/FastPak/Data/SolarPics
```

with “imagePath” being the name, and “/Users/smith/FastPak/Data/SolarPics” being the value. Likewise for the file CarPics.prop we have:

```
imagePath=/Users/smith/FastPak/Data/CarPics
```

Finally, when these two sets of name/value pairs are integrated into the application configuration properties file, the lines end up looking like this:

From SolarPics.prop:

```
INPUT_ARGS=imagePath=/Users/smith/FastPak/Data/SolarPics
```

From CarPics.prop:

```
INPUT_ARGS= imagePath=/Users/smith/FastPak/Data/CarPics
```

Input arguments passed to applications do not need to be in name value pairs!!

Any set of text after the "INPUT_ARGS=" string will be taken as command line input to an application. For example, if you had a jar application called abcd.jar and you wished to pass it the five strings "My", "name", "is", "not", "Tom", running the application manually from a terminal or console window, you would enter the command:

```
java -jar abcd.jar My name is not Tom
```

If this application had been configured to run under FastPak, then the entry for INPUT_ARGS would be as follows:

```
INPUT_ARGS= My name is not Tom
```

Information About the Applications Profiles ActiveLogo, SCSCLogo, and LogoLib

- The next set of examples (2 and 3) will involve the base code associated with the application profile named ActiveLogo. The binaries that comprise ActiveLogo, which is a flat application, are also used in the jar files associated with the profiles SCSCLogo and LogoLib. SCSCLogo and LogoLib present different ways of packaging applications for use, with or without **FastPak**. Since the binaries used to comprise all three of these profiles are identical, the user should be aware of the fact that the application can accept input parameters. Each of these applications can take one input parameter of values 480, 600, or 768 to define the size of the logo based on screen height (480 corresponds to 640 by 480, 600 corresponds to 800 by 600, and 768 corresponds to 1024 by 768) or it defaults to a value of 600. The profile associated with LogoLib uses an input parameter of 768 whereas the other two default to 600 since there is no entry associated with the name INPUT_PARAMS in their corresponding *application configuration properties files*. The thing to remember when reading the sections below is that the applications are essentially identical to one another, differing only in the way they are packaged. The application itself used by all three is comprised of only three class files, which are described as follows:

- **logo.ActiveLogo** This is the main() method for the application. It accepts a single input parameter, or none at all. If an input parameter is provided, it must be a single value of 480, 600, or 768 as described above. If the input parameter is not one of these values it will default to 600. Any parameters provided in addition to the first parameter are ignored. This main method creates a thread, passes its interpreted command line parameters to the drawing engine, launches the thread, and then immediately dies. This causes **FastPak** on some platforms to register a nearly instantaneous death on its logging GUI.
- **logo.ActiveLogo\$1** This is the result of an inner class implementing the destruction of the application.

- **SCSCLogo** This is the drawing engine that runs on its own thread. It is responsible for all the work this application does.

Example 2: Compare the file SCSCLogo.prop to ActiveLogo.prop – The *application configuration properties files* for SCSCLogo.prop and ActiveLogo.prop both define the application profiles for SCSCLogo and ActiveLogo, respectively. Both of these applications are compiled from the exact same source code, but are packaged differently. SCSCLogo is a jar application and ActiveLogo is a flat application. If the jar file associated with SCSCLogo.prop is unjarred and the manifest is read, the “Main- Class” entry in the manifest will reference the class “logo.ActiveLogo” which is the exact same entry entered for the APP_NAME entry in the *application configuration properties file* ActiveLogo.prop. The reader should compare and understand the differences in both files with respect to the property file names APP_NAME and APP_PATH and note the differences.

If the file SCSCLogo.jar is unjarred, it will contain the exact same files as found in the “FlatApps” directory underneath “logo.” An image of the manifest file used to create the jar file SCSCLogo.jar is shown below:

```
Main-class: logo.ActiveLogo
```

Figure 7 Manifest file contents for SCSCLogo.jar

The reader must read and compare the *application configuration properties file* for both SCSCLogo.prop and ActiveLogo.prop to fully grasp the difference between the two.

Example 3: Compare the file SCSCLogo.prop to LogoLib.prop - As in **Example 2** above, both of these applications are packaging variants of the source code used in the application associated with the application profile ActiveLogo. If the reader compares the properties files SCSCLogo.prop and LogoLib.prop, the following differences will be apparent:

1. LogoLib.prop has an input parameter setting of 768, whereas SCSCLogo has none. This is simply done to demonstrate the use of input parameters. If the parameter of 768 is removed, the two applications will function identically. The fact is both applications can take the exact same set of input parameters.
2. SCSCLogo has a version number assigned, whereas LogoLib has none. As mentioned previously, version numbers are provided as a convenience for developers and administrators so they can track which applications are installed on their systems, but they are completely ignored by **FastPak**.
3. SCSCLogo.prop references a file called SCSCLogo via the APP_NAME property, whereas LogoLib.prop references a file called SCSCLogoLibTest. Since both of these are designated in their properties files as being jar files (IS_JAR = true) then the actual names of the files are SCSCLogo.jar and SCSCLogoLibTest.jar.

Reviewing all the items above, it will hopefully be clear that the only significant difference is in item 3 above. So what's the difference? The application associated

with LogoLib is not a single jar file but rather a jar file that references another library in jar format, whereas SCSCLogo is a complete jar file of all class files that are generated at compile time.

FastPak internally generates and hides some of its class path settings at start up, and one of them is the directory that stores jar files. Files stored in this directory are not limited to applications, but can include libraries as well. The jar file that LogoLib.prop identifies as its application is SCSCLogoLibTest.jar. If this file is unjarred and the manifest is read, the contents of the manifest file are as follows:

```
Main-Class: logo.ActiveLogo
Class-Path: SCSCLogoLib.jar
```

Figure 8. Manifest file contents for SCSCLogoLibTest.jar

If the contents of both SCSCLogoLibTest.jar and SCSCLogoLib.jar are unjarred, they will display the contents as follows:

```
META-INF/
META-INF/MANIFEST.MF
logo/ActiveLogo$1.class
logo/ActiveLogo.class
```

Figure 9. Contents of SCSCLogoLibTest.jar

```
META-INF/
META-INF/MANIFEST.MF
logo/SCSCLogo.class
```

Figure 10. Contents of SCSCLogoLib.jar

In Figure 8 above we see the manifest that was used to create the jar file SCSCLogoLibTest.jar. As a class path it references the library file SCSCLogoLib.jar. Since **FastPak** already internally includes jar files in the directory associated with JarApps extension of the **FastPak** base directory, then this library is included by default, but the application targets only that jar library, not every jar library in the directory.

In Figure 9 above we see the actual contents of SCSCLogoLibTest.jar. It is comprised of the class files that were compiled from the source code file that contained the main() method for the application as well as the inner class used to terminate the GUI.

In Figure 10 above we see the contents of the library that SCSCLogoLibTest.jar references. The contents consist of the applications drawing engine, which does nearly all the work for the application. As a side note, in some cases it may be possible for multiple, different applications to reference the contents of a single library without the need to load them for each application. In such cases, this can conceivably reduce memory consumption and increase performance.

With that said, it's worth looking at the manifest file and actual contents of the jar file associated with the profile for SCSCLogo:

```
Main-class: logo.ActiveLogo
```

Figure 11. Manifest for SCSCLogo.jar

```
META-INF/  
META-INF/MANIFEST.MF  
logo/ActiveLogo$1.class  
logo/ActiveLogo.class  
logo/SCSCLogo.class
```

Figure 12. Contents of SCSCLogo.jar

As you can see from figures 11 and 12 above, SCSCLogo.jar is based entirely on the inclusion of all the compiled class files comprising the application instead of being based on an external library.

Example 4: Compare a remote application with its corresponding local application - *FastPak* is capable of launching applications from a web server if they are in a jar format, and, obviously, hosted by a web server. Before proceeding, please review and understand the document titled **Security Warnings Regarding *FastPak* for Java**, since launching applications using this technique can pose some security problems in some working environments.

For this particular example, the application profiles designated as “Analyzer” and “RemoteAnalyzer” will be examined. The profile for “Analyzer” refers to the application “Analyzer.jar,” which is included in the ***FastPak*** JarApps directory after installation of the ***FastPak*** kit. The other application designated by the profile “RemoteAnalyzer” uses the exact same jar file used by the “Analyzer” profile, however instead of being installed in the local installation directory, it is referencing a copy of that jar file on SCSC's own web server (www.scsc-online.com/test.) There is no difference between the jar file stored in ***FastPak's*** JarApps directory and the one stored on the server.

Security Warning: The application profile for “RemoteAnalyzer” provided in this example is intended to be used **ONLY** as an example to assist readers in understanding the difference between an application that is stored on a web server vs. one stored on a local disk. Unless security is not an issue, SCSC **STRONGLY** recommends that the binary stored on SCSC's website not be accessed, and instead a copy of the file Analyzer.jar found in the JarApps directory of ***FastPak's*** base installation directory, be copied to a local and secure network server where security can be controlled. If this is done, the properties file associated with RemoteAnalyzer will need to be modified and the **HOST_NAME** entry be changed from “www.scsc-online.com/test” to a server/path that reflects the new location of the file.

The following two listings illustrate the differences between the *application configuration properties files* for local and remotely launched applications. Figure 13 shows the contents of the *application configuration properties file* Analyzer.prop, which defines the profile Analyzer. Figure 14, shows the contents of the *application configuration properties file* RemoteAnalyzer.prop, which defines the profileRemoteAnalyzer.

```
# APP_NAME is a string - This is the actual name of the application .....
APP_NAME = Analyzer

# APP_VERSION is a string - This is a user supplied version number .....
APP_VERSION = 1.0

# APP_PATH is a string - The base path of the jar or class file that FastPak
# will be executing.
APP_PATH = /Users/smith/FastPak/JarApps/

# IS_JAR is a boolean, and will be true for jar file, false for full app
# SCSC recommends using jar files for all applications.
IS_JAR = true

# IS_REMOTE is a boolean - true = jar file from web server, false = local file
IS_REMOTE = false

# REMOTE_PATH is a server path for remote applications...leave blank for
# local apps
REMOTE_PATH =

# Hostname is the string name of the remote host, "localhost" for local files
HOSTNAME = localhost

# PRE_LOAD is a boolean - true if the app is to be pre-loaded, false if loaded
# on demand. CURRENTLY ALL APPLICATIONS ARE PRELOADED
PRE_LOAD = true

# GO_FLAG is a boolean - true if the app is to automatically execute .....
GO_FLAG = false

# INPUT_ARGS is a list of Strings - command line parameters for .....
INPUT_ARGS =

# APP_PRIORITY value sets the priority of the launch thread. Values .....
APP_PRIORITY = 5
```

Figure 13. Application Profile for Local Version using Analyzer.jar

```

# APP_NAME is a string - This is the actual name of the application .....
APP_NAME = Analyzer

# APP_VERSION is a string - This is a user supplied version number .....
APP_VERSION = 1.0

# APP_PATH is a string - The base path of the jar or class file that FastPak
# will be executing.
APP_PATH =

# IS_JAR is a boolean, and will be true for jar file, false for full app
# SCSC recommends using jar files for all applications.
IS_JAR = true

# IS_REMOTE is a boolean - true = jar file from web server, false = local file
IS_REMOTE = true

# REMOTE_PATH is a server path for remote applications...leave blank for
# local apps
REMOTE_PATH = /test

# Hostname is the string name of the remote host, "localhost" for local files
HOSTNAME = www.scsc-online.com

# PRE_LOAD is a boolean - true if the app is to be pre-loaded, false if loaded
# on demand. CURRENTLY ALL APPLICATIONS ARE PRELOADED
PRE_LOAD = true

# GO_FLAG is a boolean - true if the app is to automatically execute .....
GO_FLAG = false

# INPUT_ARGS is a list of Strings - command line parameters .....
INPUT_ARGS =

# APP_PRIORITY value sets the priority of the launch thread. Values .....
APP_PRIORITY = 5

```

Figure 14. Application Profile for Remote Version using Analyzer.jar

Note the following with regards to the two listings:

- Both listings use the same name for the application using the APP_NAME property, which is “Analyzer.”
- Both profiles have the IS_JAR value set to true. This means that when this profile is loaded, **FastPak** will be looking for a file using the APP_NAME entry (described directly above) with a “.jar” extension, hence it will both be looking for a jar file called “Analyzer.jar” and not a flat application (with a “.class” extension). *All remotely launched applications must be in a jar format.*
- The APP_PATH entry for the local listing in Figure 13 is set to the “JarApps” directory where the **FastPak** base installation placed the program. In this case, the **FastPak** installation directory is **/Users/smith/FastPak** and thus the location of the jar file **Analyzer.jar** is in the path **/Users/smith/FastPak/JarApps**. On the other hand, the APP_PATH entry for the profile associated with the remotely launched application is empty.
- The IS_REMOTE value for the locally launched application is set to false, whereas for the remotely launched application it is set to true. If this flag is true, entries for APP_PATH (defined directly above) are ignored and instead REMOTE_PATH, which will be defined below is used.
- The REMOTE_PATH value for the locally launched application is empty, whereas for the remotely launched application it's set to “/test.” If a profile has a value of IS_REMOTE set to false, then this entry will be ignored even if there is a real value in place.
- The HOST_NAME value for the local application is set to “localhost,” whereas it's set to “www.scsc-online.com” for the remote application (see the security note above regarding this entry). A “localhost” entry tells **FastPak** to look for applications under the users own system and will make use of the APP_PATH value to determine its location, whereas any other entry will attempt to connect to a web server and hunt for the associated application under its REMOTE_PATH value.

FastPak makes no real distinction between a local application and a remote application. Remote applications can be passed command line parameters just like local applications and store their associated libraries on a remote server as well. As a final note, **SCSC** cannot stress enough the importance of security when launching remote applications. If there are any doubts for a particular installation or facility, either don't use remote applications or avoid using **FastPak** with applications that require remote storage.

Using the Console and Daemon Modes

This section will cover the use of both the FastPak console and daemon modes of operation. The following paragraphs will provide instructions for configuring the script used to launch the FastPak console (which can also be used to launch FastPak in GUI or daemon mode), give an overview of commands and their syntax in console mode, and provide instructions on how FastPak in daemon mode can be started.

In the FastPak base subdirectory “lib” there is a jar application called ConsoleConfigurator.jar. This application must be run via a terminal window or command shell. It is launched as follows:

java - jar ConsoleConfigurator.jar

When the jar file is run, it will create a bash/Bourne/Korn shell compatible script on Unix, Linux, and OS X variants called *FastPakConsole*, and under Windows variants it will create a file called *FastPakConsole.bat*. Under Unix/Linux/OS X variants, this shell will not be set to be executable, but the user may feel free to do so if they so desire. Launching the script is done as follows:

Unix/Linux/OS X variants with permissions not set to executable:

sh FastPakConsole

Unix/Linux/OS X variants with permissions set to executable:

FastPakConsole

Windows variants:

FastPakConsole.bat

When the user enters the appropriate command, FastPak will start. It should be noted that although this script is intended to be used with the console, it can also launch FastPak in both GUI and daemon modes. This will be immediately noticeable to OS X users as the menu bar will no longer be used. Using this script to launch the FastPak in *any* of its modes may be useful in this mode because the user can modify class path settings, use a different version of a JVM, or modify or add JVM options to the command line.

When the console starts, it will initially display list of information related to **FastPak**. The information shown is the same as that shown under the GUI, however the console will typically scroll past it very quickly. If the window a user is using supports scrolling, a user may be able to scroll up and see this. In any case, the interactive portion of the console displays is shown in the following screen capture:

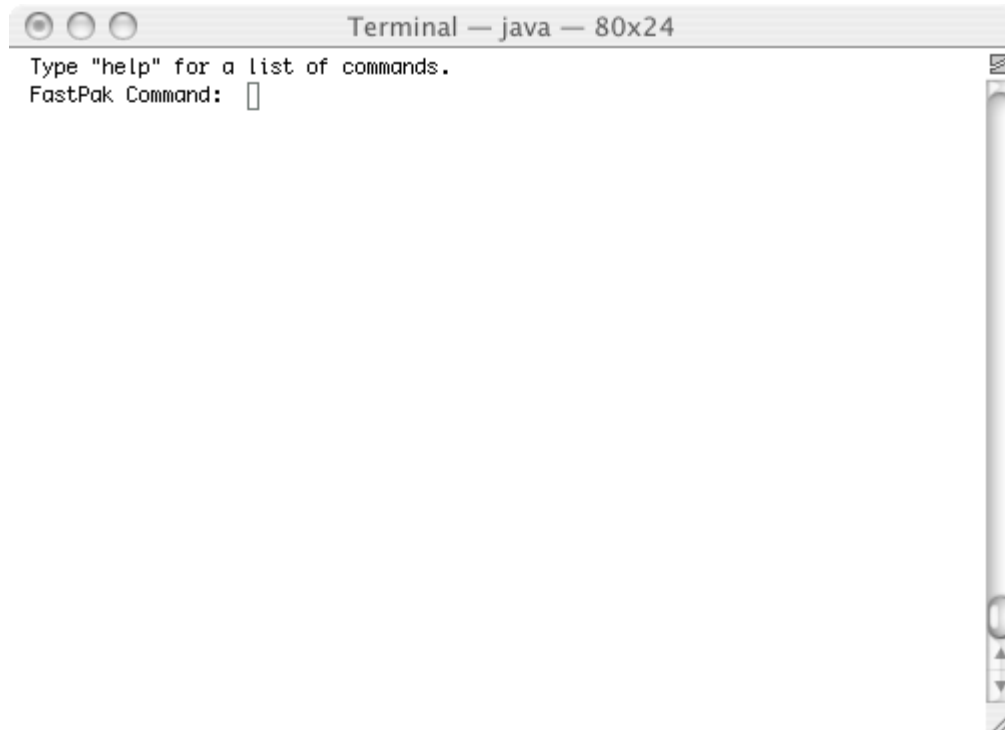


Figure 15. The FastPak consoles opening display.

All of the commands used by the console are direct implementations of the FastPak protocol. If a user types in “help” (without the double quotes), the console will display all accessible components of the protocol as follows:

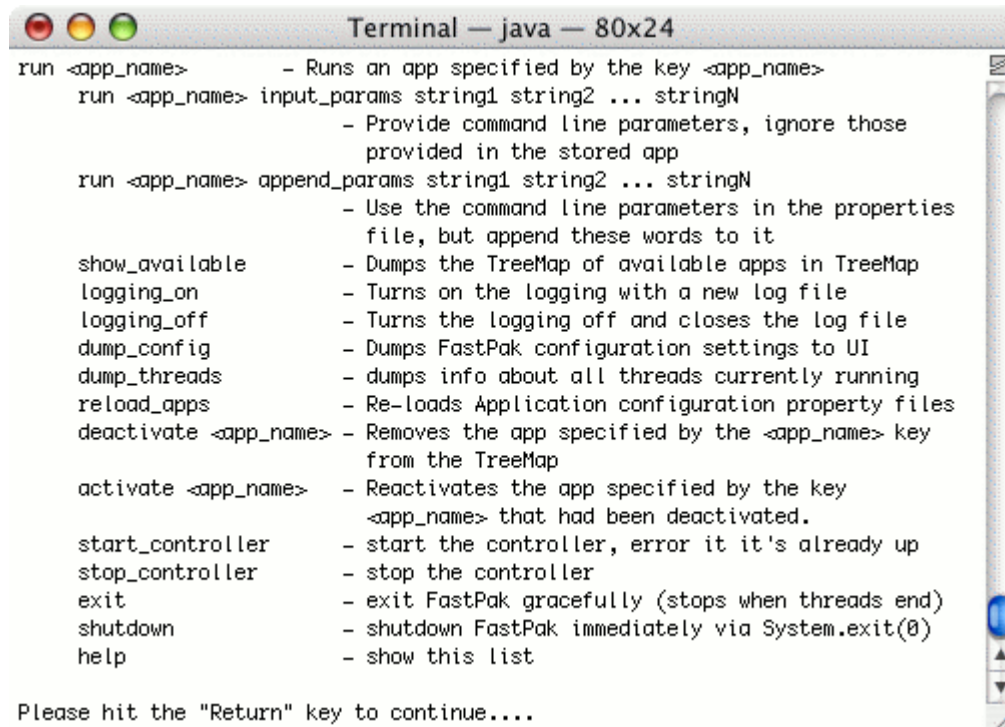


Figure 16. The FastPak consoles help display.

In the screen shot in Figure 16, the commands available to the user are on the left side, and their description is on the right with the two separated by a hyphen. The commands should be typed as shown with the exception of the angle brackets surrounding the application name.

The **FastPak** accessible components of the **FastPak** protocol are described in detail in the developers manual. The following points will hopefully make using the console a little clearer:

- The wording <app_name> actually refers to an application profile name and it doesn't include the angle brackets (<>) in actual use. For example, if you wanted to run the application defined by the profile for ActiveLogo, a user would not type in **run <ActiveLogo>**, a user would enter:

run ActiveLogo

- The first version of the “run” command shown in Figure 16 is a basic launch command, such as **run ActiveLogo**, which simply launches the application associated with the ActiveLogo profile as it's configured in its *application configuration properties file*. This includes any command line parameters defined in that profile.
- The second version of the “run” command is used to override any command line parameters provided in the profile being used. For example, the profile for “LogoLib” as initially installed uses a command line parameter of “768.” If a user wanted to launch this command from the console using a command line value of “480” instead, the syntax is:

run LogoLib input_params 480

This option should also be used if a user wants to not only override, but append parameters simultaneously. The string “input_params” is actually a tag for the **FastPak** console to mark all following text as command line overrides, hence its use is mandatory. Any string data that follows the string “input_params” will completely override any and all existing command line input that may exist in current profiles.

- The third version of the “run” command appends command line data to that defined in an existing profile. For example, the profile CarPics defines a path to the image files it uses. In this applications case, this is a mandatory field, and its associated application, ImageViewer.jar would have no idea where to look for its image files if this is not included. However, the application (as described in the regular users manual) can also accept values for a length and height. If a user wished to remap these images to a length of 320 and a height of 240, then a user could do so as follows:

run CarPics append_params length=320 height=240

This option allows the profile to launch using the location of the files defined by the `imagePath` value, but appends a length value of 320 and a height value of 240 to the original profile. The string “append_params” right after the profile name marks all text following it as command line parameters that are to be appended to the command line data that exists in the current profile.

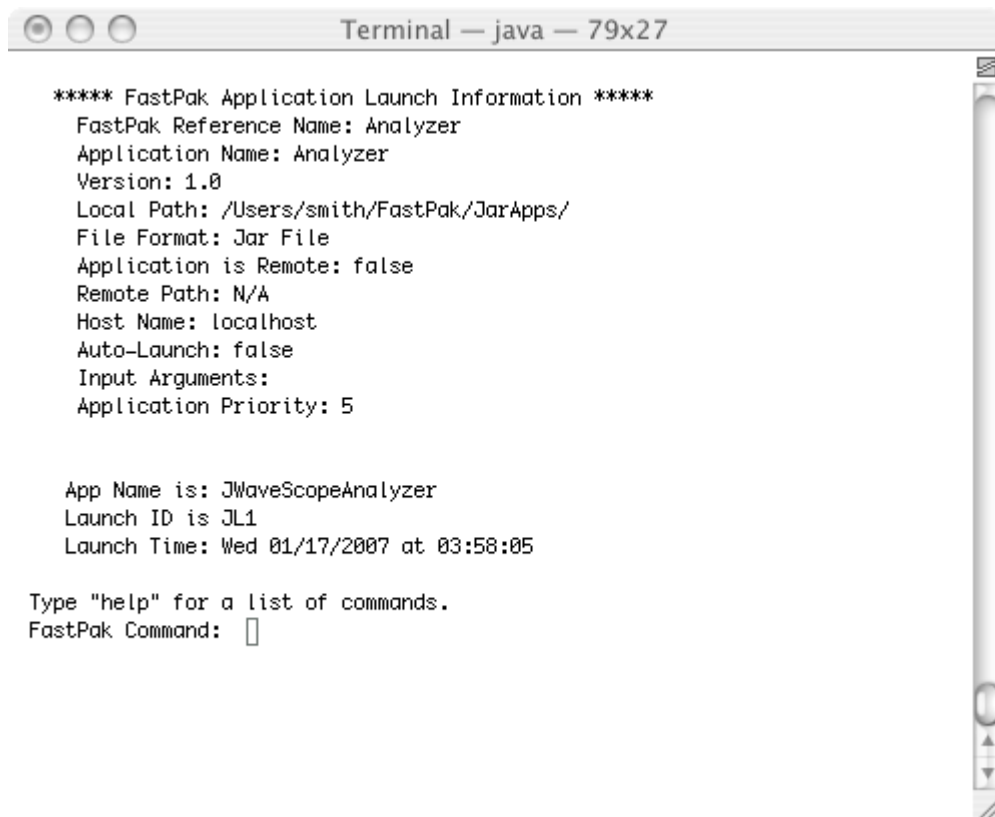
- Console and daemon modes provide two options, one to deactivate a profile, and another to reactivate a profile, that the GUI version does not. This option is really intended to be used with either a running **FastPak** instance with an active *Controller Thread* or a **FastPak** instance running in daemon mode (which has the *Controller Thread* already running). This was provided to allow an administrator to remove a profile from a system to prevent user(s) from using it, and then reactivate it at a later time. When a profile is deactivated or deactivated, the profile is simply removed from one of FastPak's internal queues that store profile information. When the profile is reactivated, it is simply put back in that list. If FastPak is stopped and restarted after a deactivation of one or more profiles, if the profiles are still available in the “AppConfig” subdirectory, then they will be available for use on startup again. In other words, deactivating an application only has a life span of the FastPak instance it's associated with. Using the profile ActiveLogo as an example, the syntax for deactivating and reactivating the profile is as follows:

deactivate ActiveLogo

reactivate ActiveLogo

- All the other commands identified in Figure 16 do exactly what the brief description above says it will, so further details on them will not be provided. Further details may be found in the developers manual under the section regarding the **FastPak** protocol.

When FastPak accepts and acts on a command line command, it will put up its response after the user enters it. For example, if a user ran the Analyzer profile via the command line, Analyzer would launch the application defined in the profile and then show the following response:

A screenshot of a terminal window titled "Terminal — java — 79x27". The window contains the following text:

```
***** FastPak Application Launch Information *****
FastPak Reference Name: Analyzer
Application Name: Analyzer
Version: 1.0
Local Path: /Users/smith/FastPak/JarApps/
File Format: Jar File
Application is Remote: false
Remote Path: N/A
Host Name: localhost
Auto-Launch: false
Input Arguments:
Application Priority: 5

App Name is: JWaveScopeAnalyzer
Launch ID is JL1
Launch Time: Wed 01/17/2007 at 03:58:05

Type "help" for a list of commands.
FastPak Command: 
```

Figure 17. The FastPak response to the command *run Analyzer*

If a users compares the output shown in Figure 17 above, it should be clear that the response given is identical to that used when using the GUI. The only difference is that at the very bottom of the screen, the console will put up a prompt for the next command where the next command can be entered.

With the console mode described, focus will now move to the daemon mode. The daemon mode may be run either by launching it via the GUI by double clicking on the **FastPak** icon, or by launching it via the same script used to launch the console in a terminal window or command shell. When the daemon mode is started, unless it has been configured to auto launch one or more applications, there may be no indication to a user that it is running (in some operating systems such as OS X, an icon will appear in the task bar indicating that it's running, but not all operating systems use this or an equivalent feature).

The daemon mode differs significantly from the console and GUI modes in that the daemon mode essentially requires that a custom interface. Without such an interface, there is really no way to control the application when it's running in this mode. For this reason, readers should review the sections on the **FastPak** protocol and the programming of the *Controller Thread* as detailed in the developers manual.