VERSION 18.0.0
APRIL 2023
702P09005

Xerox® FreeFlow® VI eCompose Software Dispatch SDK

User Guide

xerox™

Produced in the United States of America.

BR38523

Contents

Introduction

This chapter contains:

This guide is for software developers who use the FreeFlow® VI eCompose Dispatch (VIeCD) Software Development Kit (SDK) to integrate post-processing applications with the VI eCompose (VIeC) software. To use the VI eCompose software, it is recommended that you are familiar with the following software or platforms:

- Xerox® VIPP® Language
- VIeC Dispatch software
- C, C++ programming language, or the platform applications

It is recommended that users have experience with the VIeC Dispatch software. Refer to the example in the *FreeFlow® VI eCompose User Guide* and the *VI eCompose Workshop*.

An overview of VIeC internals, which includes a description of VIeCD data flow and state, is presented in this document. This information is intended to supplement the VIeCD example application and reference material contained in the *FreeFlow® VI eCompose User Guide*.

Note: All modules in the FreeFlow® VI Suite software product names have changed since the FreeFlow VI Suite 10.0 Release.

LEGACY PRODUCT NAME	NEW PRODUCT NAME
FreeFlow VI Interpreter	FreeFlow VI Compose
FreeFlow VI Interpreter Open Edition	FreeFlow VI Compose Open Edition
FreeFlow VI Designer	FreeFlow VI Design Pro
FreeFlow VI PDF Originator	FreeFlow VI eCompose
FreeFlow VIPP® Pro Publisher	FreeFlow VI Design Express

All other products not mentioned in the list keep the same name used in the previous FreeFlow VI Suite Release.

References to the VIPP® language, commands, and variable information format remain unchanged.

VI Suite Customer Forum

Xerox hosts a Community Support Forum. The VI Suite Customer forum is now part of this larger support forum, allowing you to post and review information about Xerox products and services all from one location. Please take a minute to log into this customer forum community: http://vippsupport.xerox.com.

What is the VIeCD Software Development Kit?

The VIeCD SDK consists of a collection of examples and starting points, such as source code, utilities, and libraries, which can be used to integrate VIeCD with other workflows. Most of the code provided is in C (with the single exception of the olsession example, which is in C++). One API (Application Programmer's Interface), the dispatch rule wrapper library (vtpdwrap), is presented in the class definition in the file vtpdwrap.h.

Reference documentation in HTML and Adobe PDF format is also provided. Documents in HTML format were extracted from the VIeCD SDK source code, then cross-referenced and indexed, and can be found in the /vipodsdk/docs directory of the VIeCD SDK distribution media. In addition to the PDF file you are reading, PDF documents in the form of readme files can be found in each /vipodsdk/apps subdirectory, these PDF files provide instructions for using the example applications.

Where to begin

To use the VIeCD SDK, browse to /vipodsdk/docs/index.html. This file provides the basic information you need to get started, such as:

- A brief introduction to the SDK
- Licensing information
- Where to get support, and the platforms which support the VIeCD SDK
- Building the SDK
- Assumptions about the environment and file locations
- Background about VIeCD SDK design decisions
- How the VIeCD SDK is implemented
- Descriptions of the documents and files to review to get started
- A brief description of the contents of the VIeCD SDK, including:
  - Utilities
  - Libraries
  - Example Code
  - Example VIPP® Applications
  - Win32 File Layout

Additionally, the main page, `/vipodsdk/docs/index.html`, contains the following links:

Data Structures	A page that contains a list of the data structures provided and a brief description of each data structure. To view complete descriptions of each structure, click the hypertext on this page.
File List	A page that contains a list of all documented files and a brief description of each file. To view complete descriptions of each file, clicking the hypertext on this page.
Data Fields	An indexed page that contains a list of all documented struct and union fields, and links to the structures and unions to which the fields belong.
Globals	A list of all documented functions, variables, defines, enums, and typedefs with links to their related documentation.
Examples	A list of the examples provided with the SDK, and links to the source code for each.

After you review these files, use this document for background information about, and explanations of, the files and utilities that make up the VIeCD SDK.

Documentation Overview

This user guide provides background information on the VIeCD SDK and the VIeCD In-Circuit Emulator (vtpdice). The guide is organized as follows:

Background	Provides background information on VIeCD and the VIeCD SDK. This chapter supplements the information in the *FreeFlow VI eCompose User Guide*, and adds information specific to the VIeCD SDK. The chapter provides an overview of VIeCD and the following topics: VIeCD data flow VIeCD IncomingFolders Filters Eligibility: CommandTemplates, RuleVars and the DispatchRule fieldname VIeCD job lifecycle
Examples, libraries, and utilities	Provides descriptions of the files and utilities provided with the VIeCD SDK, and examples of how to use them.
VIeC Dispatch In-Circuit Emulator	Provides an expanded description of the vtpdice utility, and includes these sections: Using vtpdice Using vtpdice in batch mode

For more information about the VIPP® Language, VI Compose, and related modules, refer to the FreeFlow® Variable Information Suite Documentation. The documentation includes the following guides:

- *FreeFlow® VI Compose User Guide*: Provides the background information required to understand and use VIPP® and applications. The guide describes the files and utilities provided with the software, the resources necessary to build VIPP® jobs, and the basics of printing with VIPP®.
- *VIPP® Language Reference Manual*: Documents the VIPP® commands, VIPP® programming tips, and error messages.
- *FreeFlow® VI eCompose User Guide*: Contains information about how to use the VI eCompose software to create and dispatch Adobe PDF documents, and administer VIeC Web servers remotely.
- *FreeFlow® VI Design Pro User Guide*
- *VIPP® Manage User Guide*
- *FreeFlow® VI Explorer User Guide*
- *FreeFlow® Variable Information Suite Documentation Glossary and Quick Reference*

For information about VIPP® training, contact a Xerox representative.

Background

This chapter contains:

VIeC Dispatch software provides a generic dispatch mechanism, which initiates, then monitors VIeC job post-processing by a customer-specified backend process such as email, fax, or a document repository. In this role, VIeC Dispatch software is considered middleware, as it mediates between completed VIeC jobs and the specified backend post-processing software.

To embed parameters and other data specific to post processing into the VIeC job, use the VIPP® BOOKMARK command.

- Parameters and other data are extracted from the field names and values of the index file generated by VIeC.
- VIeC Dispatch software transfers the parameters and other data to the specified backend software.
- Index files for each job contain the extension **.csv**.

VIeCD supports workflows that require human intervention, or a hands-off workflow, using the AutoRun feature and user filters. For example:

- Backends interface with email-disbursement systems that require:
  - Human verification or signoff of the VIeC output before the dispatch
  - Limitations on the users allowed to initiate the dispatch of such jobs
- Backends interface with a document repository that requires a hands-off workflow. The VIeC-to-VIeCD process runs without human intervention.

When VIeCD invokes a backend program, the parameters are extracted from a VIeC job index file on a line-by-line basis. The process spawns a new instance of the backend program as a new subprocess for each invocation. VIeCD does not allow any direct interaction with the backend program over stdin/stdout during that invocation. The limitation of the invocation may not be suitable for interfacing with all types of backend programs. Potential incompatibilities between the VIeC post-processing and VIeCD include:

- Programs that require some form of user or programmatic interaction in the normal mode of operation, such as a Yes or No response to a file being overwritten.
- Programs that require some form of session state over a set of transactions, such as logging in to a Microsoft Exchange server to perform email transmissions.
- Backend solutions that involve more than one discrete operation, requiring the invocation of multiple, discrete post-processing operations. Examples of the operations are concatenation or other combinations of the files specified in the DataFile Template before submission to one or more backend programs.

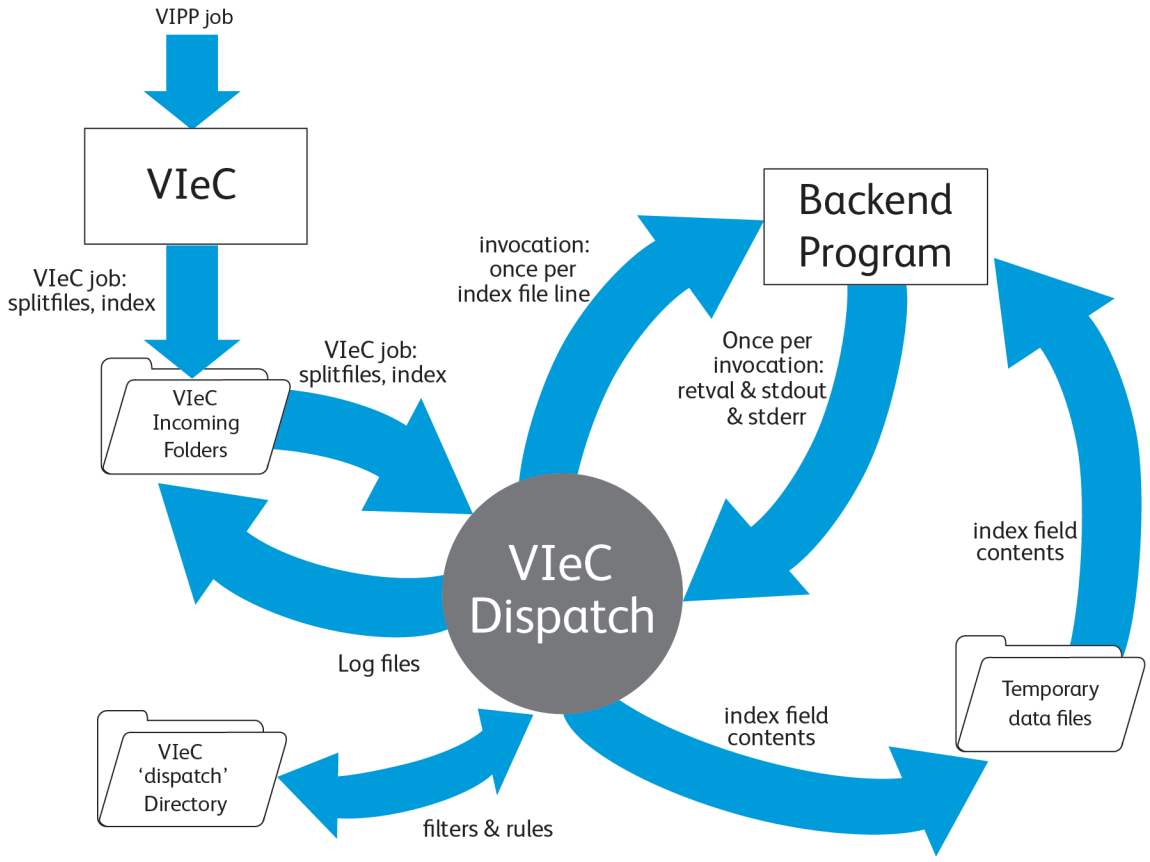
VIeCD SDK software contains example code and libraries that can provide the basis for building a shim, wrapper, proxy, or facade to resolve integration issues. The primary focus of the VIeCD SDK examples is centered on a separate server application. The server application acts as a session bridge between a simple client shim that is invoked by VIeCD on an index file line-by-line basis, and the backend program. The backend program can require human interaction or intervention, session state, or multiple post-processing operations.

An alternative to the client-server approach, for backend programs that do not require session state over a set of transactions, is to interpose a proxy program between VIeCD and the backend programs. Backend programs are invoked once for each line in the index file. The proxy assumes the responsibility for interacting with the user if necessary, and acts in the role of a facade for a collective of discrete backend programs if multiple post-processing steps are required. The proxy communicates with VIeCD through documented interfaces (retval, stdin/stdout), and with the backend programs performing the post processing. There are no explicit examples provided with the VIeCD SDK, however, you can configure the vtpdice utility to act as a proxy between VIeCD and an otherwise VIeCD-compatible backend program. It is recommended that you review the vtpdice source for possible proxy program implementation information.

For more information, refer to these sections of the *FreeFlow VI eCompose Dispatch SDK User Guide*:

- Examples
- Libraries
- Utilities
- Using vtpdice
- Using vtpdice in batch mode

VIeCD Data Flow



As VIeCD software interfaces with backend programs, VIeCD periodically inspects the VIeC IncomingFolders directories that match a specified user filter. The VIeCD software compares the field names and contents of the first line of each VIeC job index file against a repository of dispatch rules.

- If a completed VIeC job can be associated with a single dispatch rule, the job is considered eligible for dispatch and becomes a VIeCD job.
- If the VIeCD job is either manually approved for processing, or meets the user-specified criteria for a specified AutoRun filter, VIeCD sequentially resolves and applies the appropriate dispatch rule against each line of the VIeC job index file.

In the context of a dispatch rule, each of the lines of the VIeC job index file results in a separate invocation of the customer-specified backend program, effectively becoming a subtask of the VIeCD job. Sequential execution of each subtask optionally writes to disk the contents of one or more of the index file line fields, then performs an invocation of the backend program. As each subtask completes, the retval, stdout, and stderr streams are inspected in the context of the dispatch rule, to determine if a warning or error has occurred. If a warning or error occurred, the software determines if subtask processing continues or halts. The results of the subtask processing are accumulated in a log file in the same directory in which the results of VIeC job processing are stored, typically a subdirectory of the user Incoming directory.

VIeCD IncomingFolders Filters

VIeCD monitors the VIeC IncomingFolders directory in the context of a specified IncomingFolders filter. The default setting allows VIeCD to monitor all IncomingFolders for all VIeC users. However, VIeCD can be set to process jobs for a particular set of users, and/or for a particular set of IncomingFolders, as only the specified IncomingFolders of the VIeC users that match the specified user filter are considered for VIeCD processing.

Eligibility: CommandTemplates, RuleVars, and the DispatchRule FieldName

VIeC Dispatch determines the eligibility for processing of completed VIeC jobs that match the IncomingFolders filter. If the VIeC job can be matched to a single dispatch rule, VIeC Dispatch deems the job eligible for processing. The section of a dispatch rule that determines which back-end program is invoked, and with which parameters, is the CommandTemplate.

INDEX FILE

To determine eligibility, VIeC Dispatch software inspects the index file field names of each VIeC job, then compares the fields against each CommandTemplate section of the available dispatch rules.

In Example 1, if no dispatch rule CommandTemplate contains the mailto field, the VIeC job is ineligible because no dispatch rule can be applied. If the index file field names of a VIeC job are matched with the CommandTemplate of exactly one dispatch rule, the file is considered eligible.

Example 1

In this example, the following VIeC job index file field names, and single dispatch rule CommandTemplate are:

```
...,"Pages","FileSequence","mailto"
```

```
blat c:\bodytemp.txt -t $mailto
```

The VIeC job is eligible for VIeCD processing if the dispatch rule in the example is the only one with a CommandTemplate containing the single mailto field.

Alternately, the VIeC job is ineligible for processing if there is another rule containing the following CommandTemplate variable information, because more than one dispatch rule can be applied:

```
splat -x $mailto
```

Example 2

This example assumes:

- Two VIeC jobs, each containing one of these index file field names:

  ```
  ...,"Pages","FileSequence","mailto"
  ```

  ```
  ...,"Pages","FileSequence","mailto","cc"
  ```

- Exactly two dispatch rules, each containing one of the following as the CommandTemplate:

  ```
  blat c:\bodytemp.txt -t $mailto
  ```

  ```
  blat c:\bodytemp.txt -t $mailto -c $cc
  ```

In this example, both VIeC jobs are eligible for processing. VIeCD recognizes that the first VIeC job cannot be used with the second dispatch rule, because there is no field named cc in the first job index file. The first dispatch rule is applied for the first VIeC job. VIeCD recognizes that the second VIeC job cannot be used with the first dispatch rule, because there is no field named cc in the first dispatch rule CommandTemplate. The second dispatch rule is applied to the second VIeC job.

RULEVARS

In addition to the index file field names defined by a VIeC job, a dispatch rule can define additional fields and values to apply to the CommandTemplate. A typical requirement for this is a dispatch rule whose CommandTemplate requires a password to execute. It is not desirable to include the password in the VIeC job. You can specify additional fields and values in the RuleVar section of the dispatch rule.

Example 3

In this example, the dispatch rule CommandTemplate and the single dispatch rule with the user field present are as follows:

```
foo -user $user -password $password
```

```
...,"Pages","FileSequence","user"
```

In this event, the VIeC job is ineligible for processing, because there are no password fields among the index file field names.

Example 4

The VIeC job in Example 3 becomes eligible for processing when the dispatch rule has a RuleVar entry for the password field:

```
password=mypassword
```

In this event, the password field of the CommandTemplate is supplied by the dispatch rule RuleVar definition.

RESERVED INDEX FILE FIELD NAME

The final mechanism used in VIeC Dispatch for choosing among otherwise ambiguous dispatch rules for a particular VIeC job, involves the use of a reserved index file field name. If among the index file field names the VIeC job has the reserved field name DispatchRule, for the first line of the index file, the content of that field is matched against the rule names of the set of dispatch rules that are ambiguous.

Example 5

Ambiguity is resolved and the dispatch rule is applied to the VIeC job, when a job with the index file field names of mailto and DispatchRule meets the following conditions:

- The job is compared to the two rules containing blat and splat, described in Example 1, because of the mailto fieldname.
- The contents of the reserved field name, DispatchRule, for the first line of the index record is SMTP email
- One of the dispatch rules has SMTP email as its dispatch rule name

AutoRun Filters

Once a VIeC job is deemed eligible for processing by VIeCD, it is compared to the currently specified AutoRun filter. AutoRun filters allow the specification of a particular set of VIeC users and a particular set of IncomingFolders for those users, to determine which eligible VIeC jobs are automatically processed by VIeCD. The default setting is to disallow AutoRun for all users, requiring instead that each job be selected manually for processing.

Processing

When VIeC Dispatch processes a VIeC job, it reads the index file for the VIeC job one line at a time, and applies its field values in the context of the applicable dispatch rule. Each application results in a separate invocation of the backend program specified in the dispatch rule's CommandTemplate. E ach of these invocations (one for each line in the VIeC job index file) is considered a subtask of the job.

The following will occur for each line of the index file, until either a halt-on-warning or halt-on-error condition is detected, or there are no lines left in the index file to process:

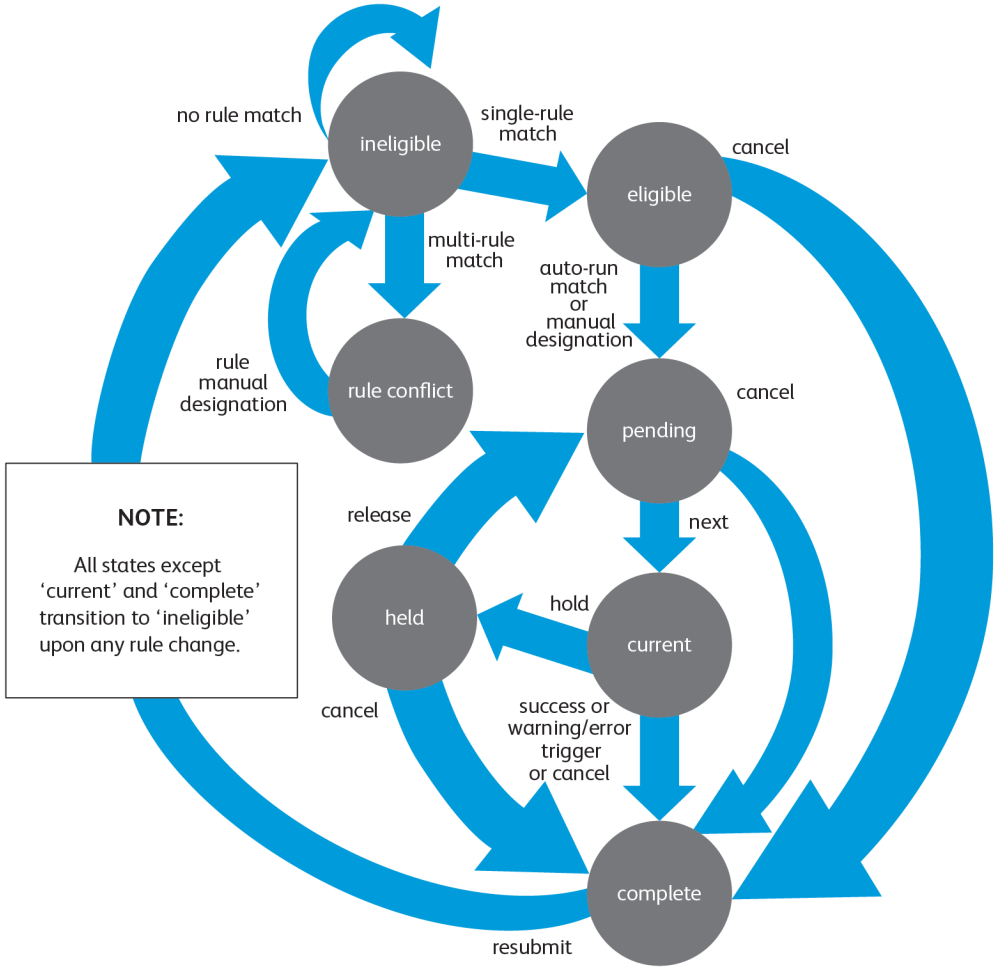
- If any of the field names are present in the DatafileTemplate section of the dispatch rule, their contents are written to disk as specified. The file(s) written are considered temporary, and their contents are considered valid only for the duration of the particular invocation of the backend program specified in the CommandTemplate.
- The field names in the CommandTemplate are then resolved by substituting their values from the RuleVars (if any) and from the values extracted from the pertinent line of the index file. The result is the command and parameters intended for the invocation of the backend program. The command and parameters are then submitted to the operating system for execution as a subprocess of VIeC Dispatch.
- When the subprocess for the backend program completes, VIeCD interprets its retval, stdout and stderr streams in the context of the pertinent sections of the dispatch rule to determine whether a warning or error situation has been triggered, and if so whether processing of the remaining lines of the index file should continue.

VIeCD Job Lifecycle

During its lifecycle, a VIeCD job can transition between any of these states:

- ineligible
- rule conflict
- eligible
- pending
- current
- held
- complete

These states are reflected in the Current Job and tabbed sections of the main VIeC Dispatch GUI. These sections are described in more detail as follows:



INELIGIBLE

All VIeC jobs that make it past the IncomingFolders filter result in the instantiation of a VIeCD job with an initial state of ineligible, creating the start of the VIeCD job lifecycle. VIeCD jobs entering this state are tested immediately for eligibility when the first index file record field names are evaluated against the CommandTemplate and RuleVar of each available dispatch rule.

One (and only one) dispatch rule

The VIeCD job transitions to the eligible state.

More than one dispatch rule

The job transitions to the rule-conflict state.

No dispatch rules

The job remains in the ineligible state.

RULE CONFLICT

Jobs in this state are those that have more than one applicable dispatch rule. Transition out of this state will occur if:

- The rule to be applied is manually selected.
- Any dispatch rule is changed or added.

In either case, the job will transition back to the ineligible state where it will then be re-evaluated. Once returned to the ineligible state:

- The job will transition to the eligible state if a dispatch rule was manually selected.
- The job may remain ineligible if no dispatch rules can be applied.
- The job may transition back to rule conflict if more than one applicable dispatch rule remains.

ELIGIBLE

Jobs in this state are eligible for processing but are not yet pending. A job remains in an eligible state until it is manually selected by the user for processing or meets the criteria of the current AutoRun filter. When the job meets the AutoRun filter criteria it then automatically transitions to the pending state.

Canceling a job in an eligible state causes it to transition to a complete state with a status of cancel.

PENDING

Jobs in this state will be processed in the order shown in the VIeC Dispatch Pending display, topmost first. The order of the pending jobs can be changed using the GUI.

Canceling a job in a pending state causes it to transition to a complete state with a status of cancel.

CURRENT

A job in this state is being processed. There can be only one job in this state.

Holding a current job causes it to temporarily suspend processing (in a graceful manner) and move into the held state.

Canceling a job in a current state causes it to transition to a complete state with a status of cancel.

HELD

A held job has suspended processing. Releasing the job causes it to transition to a pending state to await further processing.

Canceling a job in a held state causes it to transition to a complete state with a status of cancel.

COMPLETE

This is the end state of the VIeCD job lifecycle. Completed jobs have one of these statuses:

- success
- warning
- failure
- cancel

The resubmit transition is supplied for convenience. Resubmitting a job clears all accumulated state, tracking, etc., for that job and returns it to the ineligible state, allowing it to be reprocessed as if it had just come into the system for the first time.

Examples, Libraries, and Utilities

This chapter contains:

The VIeCD SDK provides examples, libraries, and utilities, which can be used to integrate backend applications with VI eCompose. Descriptions of these components are provided here.

Examples

These code examples are provided with the VIeCD SDK:

forward

Simple file forwarding for VIeCD.

client

Simple socket client that illustrates connecting to a VIeCD session application.

server

Simple socket server that you can use as the basis for a VIeCD back-end session application.

server2

More sophisticated socket server that you can use as the basis for a VIeCD back-end session application. The example shows how to use the `vtpdsession` library.

olsend

Socket client that connects to a Microsoft Outlook session bridge.

olsession

Socket server acting as a VIeCD back-end session bridge to MS Outlook.

wrap

Shows how to use the dispatch rule wrapper library `vtpdwrap`.

You can find the source for the examples in the `src/examples` section of the *VIeCD SDK* distribution. The pre-compiled binaries for the examples are in the bin section. Sample VIPP® applications that you can use to exercise the examples are in the apps section.

The examples are described in more detail in the sections that follow.

FORWARD

The most direct way to resolve an incompatibility between VIeCD and a back-end program, is to replace the back-end program with one more programs that are compatible with VIeCD. For example, a simple use of VIeCD is to automate the copying of the PDF files from completed VIeC jobs to another location, which is known as job forwarding. One obvious way to automate job forwarding, is to place an invocation of the Windows command line utility xcopy in the CommandTemplate of a dispatch rule. However, to copy a file to a new location, xcopy prompts for confirmation, if copying the file overwrites another file. Prompting is incompatible with VIeCD, because no interaction with the subprocess can occur over stdin or stdout, and causes VIeCD to hang or time out. Additionally, the return value from xcopy is not informative if other problems are encountered during processing.

The forward example shows one way to wrap functionality to be more compatible with the VIeCD invocation environment. The program is not interactive. Instead, the program has command-line arguments that specify whether to overwrite a file, if a file exists. The return values indicate the following:

0	success
negative values	specific usage or environment error
positive return values	system error, such as disk full, or permissions error

Understandably, it is practical to replace only the simplest back-end processes in this way.

For instructions on how to use the forward example, refer to `/vipodsdk/apps/forward/readme.pdf`.

CLIENT

The client example provides a simple shim between VIeCD and a server, in which the client communicates with the sever over a socket connection. The server is assumed to be up and running before starting the VIeCD job communication using the client. The client example will communicate with either the server or server2 example servers.

If the client is called out in the `CommandTemplate` of a dispatch rule, each time it is invoked or spawned as a separate process by VIeCD:

- The client establishes a connection with the server and passes to the server the parameters with which it was invoked.
- The client then waits for the server to provide the return value that represents the status of the server processing of that particular transaction.
- The return value from the server is passed back to VIeCD by the client as the client return value.

Refer to `/vipodsdk/apps/client/readme.pdf` for instructions on how to use the client example.

SERVER

This example provides a rudimentary server that communicates with the client example described above. It does not actually do any work upon receiving a transaction request from a client process, and always returns the same canned return value to represent the transaction status.

The implementation is intentionally a minimalist one, as the server only bounces back return values to each incoming client connection. There is no way to signal the server process to exit as it must be manually terminated. Upon termination the server makes no allowance for the completion of any pending client connections before it exits. The server shuts down, and lets client connections resolve themselves.

While the server example can be extended to perform useful work and enhanced to be more robust, its main purpose is to stand in contrast to the `server2` example below.

Refer to `/vipodsdk/apps/server/readme.pdf` for instructions on how to use the server example.

SERVER2

This example is equivalent functionally to the server example above. However, it illustrates how to leverage the `vtpdsession` library to produce the same functionality in much less user-level code. The code is also more robust, as the `vtpdsession` library implementation is much more conscientious about allowing connected clients to exit as part of its cleanup procedure.

Refer to `/vipodsdk/apps/server2/readme.pdf` for instructions on how to use the server2 example.

OLSEND

Based on the previous client example, `olsend` performs in a similar fashion, but in this case as a shim between VIeCD and the `olsession` example that follows.

Refer to `/vipodsdk/apps/olsend/readme.pdf` for instructions on how to use the `olsend` example.

OLSESSION

With the server2 example as a starting point, olsession is a server that establishes a login session with Microsoft Outlook. The olsession server acts as a session bridge, which has the following features:

- Allows VIeCD to interface with a program, in this case Outlook, that can require user interaction. If Outlook is not already running when the server is started, a login and password are required.
- Benefits from session state. Session state is a login, followed by multiple transactions, then a logout on exit.
- Cannot interact directly with VIeCD because the VIeCD does not have a retval and does not use stdin or stdout.

The olsend client, in combination with the olsession server, provides a functional VIeCD to Outlook solution suitable for real-world use, similar to how the public-domain blat utility can be used to interface VIeCD with SMTP/POP3 email servers. For more information on the blat utility, refer to http://www.interlog.com/~tcharron/blat. html.

In addition, the olsession implementation demonstrates:

- How to build on the VIeCD SDK client and server2 examples to interface VIeCD with a fairly sophisticated backend program and in this case it is Microsoft Outlook.
- That the VIeCD SDK components are compatible with C++ programs. The olsession code is C++, linked with the vtpdsession and OOC libraries, which are written in ANSI C.
- That VIeCD can be integrated with Microsoft applications through their Component Object Model (COM) interface. Most high-end Microsoft applications have programmatic interfaces that can be accessed through COM.

For instructions on how to use the olsession example, refer to `/vipodsdk/apps/olsession/readme.pdf`.

WRAP

The wrap example shows how to use the `vtpdwrap` library to read, alter, and write a dispatch rule.

This example also shows how to notify VIeCD that a dispatch rule has been programmatically altered. When a dispatch rule has been added, changed, or deleted from the VIeCD environment, VIeCD must re-evaluate all completed VIeC jobs and unprocessed VIeCD jobs in terms of their eligibility for VIeCD processing, in the context of the new rule set. When rule changes are made from within the VIeCD GUI, VIeCD knows to do this automatically since it was the source of the change. However, when a rule change occurs due to the actions of an external program, VIeCD must be notified as shown in the wrap example.

Refer to `/vipodsdk/apps/wrap/readme.pdf` for instructions on how to use the wrap example.

Libraries

The libraries provided with the VIeCD SDK can be used to generate dispatch rules from within applications or other front-end processes, and to implement session server code.

VTPDWRAP

`vtpdwrap` is a wrapper library that can be used to generate, read, alter, or write VIeC dispatch rules. Use of the `vtpdwrap` library is strongly encouraged over the direct manipulation of dispatch rules, as it insulates the solution provider from the effects of any future changes to the underlying dispatch rule file format.

VTPDSESSION

Some of the examples provided with the VIeCD SDK illustrate how to establish a bridge with a back-end process, or a process that requires some form of state over a set of transactions. These examples are implemented using a socket-based client/server model. The implementation of the underlying session server code is fairly generic, and has been extracted into its own library, `vtpdsession`. This library is used by the `server2` and `olsend` examples, and can also be used by solution providers if a socket-based approach is appropriate for their session bridge implementations.

Utilities

For development and integration activities, it is useful to have a way of monitoring the communication between VIeCD and the back-end program. To support this communication, the VIeCD In-Circuit Emulator (vtpdice) utility is provided with the VIeCD SDK. vtpdice can be interposed between VIeCD and a back-end program to provide diagnostic information and automatic fault insertion, to help in hardening during unit testing. vtpdice is a useful utility, and is an illustrative example of how to interpose a proxy between VIeCD and a back-end program. The full source code for vtpdice is included with the VIeCD SDK.

For more information, refer to VIeC Dispatch In-Circuit Emulator.

VIeC Dispatch In-Circuit Emulator

This chapter contains:

The VIeC Dispatch In-Circuit Emulator (vtpdice) is intended to provide a test harness and diagnostic tool for use during VIeCD back-end plug-in development and integration.

Note: The VIeC Dispatch In-Circuit Emulator (VIeCDICE) utility is found in the SDK as vtpdice.exe.

vtpdice is intended to be invoked as a back-end process by VIeCD in the same way as a third-party OEM back-end program is invoked. More specifically, the intent is for VIeCD to invoke vtpdice with the same arguments that otherwise are passed to an arbitrary target back-end program.

For example, suppose the intent is to target Blat, which is a freeware command-line POP3 email submission tool, from VIeCD. During development, it can be useful to get a trace of what Blat receives before invoking blat in the final program. In this case, VIeCD can invoke vtpdice instead of Blat, but with the same arguments as if Blat is invoked.

Invoking vtpdice can be useful, because you can configure vtpdice to respond in four different ways:

Use Case 1	Log the arguments passed, and return a specified return value to VIeCD.
Use Case 2	Log the arguments passed, and emit the contents of specified files to either stdout, stderr, or both, and return a specified return value to VIeCD.
Use Case 3	Track the number of times vtpdice has been invoked, with a normal response as described in Use Case 2. After a specified number of invocations have been performed, insert a fault in the response stream and return a specified fault return value to VIeCD.
Use Case 4	Act as a proxy for the real target backend program, such as blat. In this configuration, VIeCD calls vtpdice, and vtpdice calls the target program. vtpdice intercepts and logs the stdout, stderr, and retval of the target program, and feeds them back to VIeCD.

Using vtpdice

At present vtpdice can be used only on Windows systems.

vtpdice is intended to be transparent with regard to the arguments passed to it, compared to the arguments potentially passed to a target program. This means that meta-test information and configuration cannot be passed to vtpdice using the command line of vtpdice. As a consequence:

- Each vtpdice test has its own unique directory.
- Each test directory must contain at a minimum a file named `vtpdice.tst`.
- vtpdice reads `vtpdice.tst` for the configuration information of that test run.
- vtpdice appends all logged information to a file `vtpdice.log` in the test directory, unless an environment variable `vtpdice_test_log` exists. In this case, all logged information is appended to the file specified in that environment variable.
- After each invocation, vtpdice increments the number of times it has been invoked to the file `vtpdice.rct` to keep track of the record count. This is number of times vtpdice has been invoked in the context of that test directory. vtpdice tracks this number to know when to fault after a particular number of invocations if that is part of the test invocation. To reset the number of records, delete this file.

Upon invocation, vtpdice examines the contents of the environment variable `vtpdice_test_dir`. If that environment variable exists, it uses the contents as the path of the directory to use for that test run. If that environment variable does not exist, it looks in its current working directory for a file called `vtpdice.ini`. If that file exists, it uses the contents as the path of the test directory. If neither the environment variable nor the `vtpdice.ini` file exist, or the test directory specified within them does not exist, the test run is aborted.

If the test directory path is a relative path, it is considered relative to the vtpdice current working directory.

Given a valid test directory path, vtpdice opens and reads the `vtpdice.tst` file in that directory. If the file does not exist, the test is aborted.

Use Cases 1–4 are descriptions of how to configure the `vtpdice.tst` file contents to get the desired test behavior from vtpdice. For the cases that follow, assume:

- **vtpdice.exe** is in directory `C:\vtpdice`
- A test directory `C:\vtpdice\test`
- A test configuration file `C:\vtpdice\test\vtpdice.tst`
- A file `C:\vtpdice\vtpdice.ini` that contains the single line:

  ```
  test
  ```

USE CASE 1

Use Case 1 logs the arguments passed, and returns a specified return value to VIeCD.

`vtpdice.tst` contents:

```
retvalNormal:0
```

Invoke vtpdice:

```
vtpdice a b c
```

`vtpdice.log` contents:

```
Test run #0, time: Fri Jun 13 13:12:23 2003
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 'a'
argv[2]: 'b'
argv[3]: 'c'
No stdout source file specified, nothing sent to stdout.
No stderr source file specified, nothing sent to stderr.
Value returned: 0
```

USE CASE 2

Use Case 2 logs the arguments passed, and emits the contents of specified files to either stdout, stderr, or both, as well as returning a specified return value to VIeCD.

`vtpdice.tst` contents:

```
retvalNormal:0
stdoutSourcePathNormal:stdoutNorma1.txt
stderrSourcePathNormal:stderrNorma1.txt
```

Relative paths specified in a `vtpdice.tst` file are relative to the test directory, not the vtpdice current working directory. Two files, `stdoutNormal.txt` and `stderrNormal.txt`, containing information to be sent to stdout and stderr as a part of the test, must be in the test directory.

Invoke vtpdice:

```
vtpdice d e f
```

`vtpdice.log` contents:

```
Test run #0, time: Fri Jun 13 13:12:23 2003
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 'd'
argv[2]: 'e'
argv[3]: 'f'
```

Contents of file `C:\vtpdice\test\stdoutNormal.txt` sent to stdout, as follows:

```
--- start stdout stream ---
'Normal' stdout contents goes here...
--- end stdout stream ---
```

Contents of file `C:\vtpdice\test\stderrNormal.txt` sent to stderr, as follows:

```
--- start stderr stream ---
'Normal' stderr contents goes here...
--- end stderr stream ---
Value returned: 0
```

USE CASE 3

Use Case 3 tracks the number of times vtpdice was invoked with a normal response as in Use Case 2. However, after a specified number of invocations are performed, a fault is inserted in the response stream. The contents of specified fault files is sent to stdout, stderr, or both, and returns a specified fault-return value to VIeCD. Subsequent invocations return the normal stdout/stderr/retval behavior.

In this example, vtpdice inserts a fault at record 2, the third record, because vtpdice counts from zero. At the fault, vtpdice returns a value of `-1`, and different information on stdout/stderr.

`vtpdice.tst` contents:

```
retvalNormal:0
stdoutSourcePathNormal:stdoutNormal.txt
stderrSourcePathNormal:stderrNormal.txt
faultRecord:2
retvalFault:-1
stdoutSourcePathFault:stdoutFault.txt
stderrSourcePathFault:stderrFault.txt
```

To reset the record count to zero, delete vtpdice.rct, if the parameter exists.

Invoke vtpdice:

Invoke vtpdice four times:

```
vtpdice g h i
vtpdice j k l
vtpdice m n o
vtpdice p q r
```

`vtpdice.log` contents:

```
Test run #0, time: Fri Jun 13 13:12:23 2003
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 'g'
argv[2]: 'h'
argv[3]: 'i'
```

Contents of file `'C:\vtpdice\test\stdoutNormal.txt'` sent to stdout, as follows:

```
--- start stdout stream ---
'Normal' stdout contents goes here...
--- end stdout stream ---
```

Contents of file `'C:\vtpdice\test\stderrNormal.txt'` sent to stderr, as follows:

```
--- start stderr stream ---
'Normal' stderr contents goes here...
--- end stderr stream ---
Value returned: 0
Test run #1, time: Fri Jun 13 13:12:23 2003
```

```
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 'j'
argv[2]: 'k'
argv[3]: 'l'
```

Contents of file 'C:\vtpdice\test\stdoutNormal.txt' sent to stdout, as follows:

```
--- start stdout stream ---
'Normal' stdout contents goes here...
--- end stdout stream ---
```

Contents of file 'C:\vtpdice\test\stderrNormal.txt' sent to stderr, as follows:

```
--- start stderr stream ---
'Normal' stderr contents goes here...
--- end stderr stream ---
Value returned: 0
Test run #2, time: Fri Jun 13 13:12:23 2003
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 'm'
argv[2]: 'n'
argv[3]: 'o'
FAULT INSERTED at record #2:
```

Contents of file 'C:\vtpdice\test\stdoutFault.txt' sent to stdout, as follows:

```
--- start stdout stream ---
'Fault' stdout contents goes here...
--- end stdout stream ---
```

Contents of file 'C:\vtpdice\test\stderrFault.txt' sent to stderr, as follows:

```
--- start stderr stream ---
'Fault' stderr contents goes here...
--- end stderr stream ---
Value returned: -1
Test run #3, time: Fri Jun 13 13:12:23 2003
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 'p
argv[2]: 'q'
argv[3]: 'r'
```

Contents of file 'C:\vtpdice\test\stdoutNormal.txt' sent to stdout, as follows:

```
--- start stdout stream ---
'Normal' stdout contents goes here...
--- end stdout stream ---
```

Contents of file `'C:\vtpdice\test\stderrNormal.txt'` sent to stderr, as follows:

```
--- start stderr stream ---
'Normal' stderr contents goes here...
--- end stderr stream ---
Value returned: 0
```

USE CASE 4

Use Case 4 acts as a proxy for the real target back-end program. In this configuration, VIeCD calls vtpdice, and vtpdice calls the target program. vtpdice intercepts and logs the target program's stdout, stderr and retval, and also feeds these back to VIeCD. So to VIeCD, it looks as if the target program was directly invoked and actually performed the work. In this fashion, vtpdice acts as a proxy or shim that can be inserted between VIeCD and any target back-end program for tracing or other diagnostic purposes, and perhaps even in the field to capture information for troubleshooting customer problems.

The VIeCD SDK contains a simple loop-back program that tests this use case. The loop-back program serves as an example application invoked by vtpdice.

`vtpdice.tst` contents:

```
pathToProxyFor:../../../bin/loopback.exe
```

This path can be either relative or absolute. If relative, it is relative to the test directory, not the vtpdice current working directory.

Invoke vtpdice:

```
vtpdice s t u
```

vtpdice.log contents:

```
Test run #0, time: Fri Jun 13 13:12:23 2003
Invocation argc, argv:
argv[0]: 'vtpdice'
argv[1]: 's'
argv[2]: 't'
argv[3]: 'u'
Performing as proxy for: 'C:\vtpdice\test\../../../bin/loopback.exe'
Arguments passed:
argv[0]: 'C:\vtpdice\test\../../../bin/loopback.exe'
argv[1]: 's'
argv[2]: 't'
argv[3]: 'u'
--- start received stdout stream ---
Hello, I am the program located at
'C:\vtpdice\test\../../../bin/loopback.exe'.
I was called with 3 arguments, as follows:argv[1]: 's'argv[2]:
't'argv[3]: 'u'End of argument list.
Sending 'Hello, stderr!' to stderr...
And now sending '123' as a return value...bye!
--- end received stdout stream ---
```

```
--- start received stderr stream ---
Hello, stderr!
--- end received stderr stream ---
Value returned: 123
```

Using vtpdice in Batch Mode

vtpdice can be run in batch mode from the command line. Running in batch mode from the command line allows you to exercise vtpdice and any specified end-user back-end programs to ensure that the tests themselves perform as expected before you run live diagnostic jobs through VIeCD. You can use batch mode to perform a make-ready verification to ensure that the tests are functioning properly before they are invoked by VIeCD and vtpdice.

By setting or adjusting the *vtpdice_test_dir* and *vtpdice_test_log* environment variables, you can arrange for a number of different tests to be run as a batch.

Look for the file `vipodsdk/bin/runtests.bat`, the result of running the batch test will appear in the file `alltest.log`. Examining the batch file, the test directories, located in `vipodsdk\src\examples`, and the test directory contents provides details on the use of vtpdice in non-batch mode as well as batch mode.

To run the batch test yourself, change the absolute path for the environment variables to match the `vipodsdk` location.