TWAIN Direct™ Specification: User's Guide

Ratified October 2nd 2017
Revision 1.0

History

Date	Version	Comment
September 15th, 2017	1.00	First version

Notes

Notes
• (none)

Contents

Glossary of Terms

This section establishes the meaning of words used within the Specification.

Word	Meaning
action	A TWAIN Direct command (e.g. “configure”).
application	A program that sends TWAIN Direct commands to a scanner.
attribute	A configurable item, such as compression, resolution, etc.
communication manager	A system that discovers scanners, registers them and provides cloud and/or local area net communication channels.
exception	A TWAIN Direct directive that changes the way a TWAIN Direct task is evaluated by a scanner, when it cannot exactly match a specific request within a task.
JSON	A lightweight data-interchange format.
pixelFormat	The combination of a color space and a bit depth, for instance, rgb24 indicates a color image with 24 bits of depth.
scanner	Any device that captures images for an application.
source	A physical provider of images, such as a flatbed or an automatic document feeder.
stream	A collection of one or more sources, which combined together results in a stream of images during scanning.
task	A TWAIN Direct construct used to issue actions to a scanner.
topology	The combination in a configure action of a stream, source and pixelFormat used to address components within the scanner.
user	A person in control of an application and a scanner.

References

This section lists standards, guides and resources cited in this document.

Word	Meaning
Base64	RFC 3548: The Base16, Base32, and Base64 Data Encodings https://tools.ietf.org/html/rfc4648
Google JSON Style Guide	Google JSON Style Guide https://google-styleguide.googlecode.com/svn/trunk/jsoncstyleguide.xml
JavaScript Reserved Words	List of reserved words http://www.w3schools.com/js/js_reserved.asp
JSON	ECMA-404 The JSON Data Interchange Standard http://json.org
PDF/raster	PDF Raster Documents http://pdfraster.org
TWAIN Direct	Website for TWAIN Direct http://twaindirect.org
TWAIN Direct Sample Code	Repository for TWAIN Direct sample code https://github.com/twain/twain-direct
TWAIN Direct UUID Version 1	**211a1e90-11e1-11e5-9493-1697f925ec7b** https://www.uuidgenerator.net (generation source)
UUID	A Universally Unique IDentifier (UUID) URN Namespace http://www.ietf.org/rfc/rfc4122.txt TWAIN Direct UUID strings must be represented as lowercase hexadecimal values without curly brackets. Dashes separate the hexadecimal values (the numbers represent the number of hexits in each section): 8-4-4-4-12

Reading the TWAIN Direct Documents

Audience

These documents should be read by both application developers and scanner vendors. They are not intended for end users.

TWAIN Direct focuses primarily on application developers. Applications represent the needs of the end users, and since there are considerably more application writers than scanner vendors, improving the development experience in this area offers the greatest potential for influencing the market.

The complete TWAIN Direct Specification is divided into five main documents, which are subdivided into chapters.

TWAIN Direct

The specification provides an overview of the entire standard, and goes into detail on the TWAIN Direct task language and metadata.

- The User's Guide (this chapter) should be read completely to gain an understanding of TWAIN Direct's goals, and how the RESTful API, tasks, metadata and PDF/raster are used to achieve them.
- The chapter on Tasks is a reference guide to the task elements defined by this specification. Custom task elements may be described by individual scanner vendors, contact them for more information.
- The chapter on Metadata is a reference guide to the standard metadata elements defined by this specification. Custom task elements may be described by individual scanner vendors, contact them for more information.

TWAIN Local

Covers scanner enumeration on the local area network (LAN), and the RESTful API protocol used to communicate with a scanner.

- The mDNS and DNS-SD chapter describes how scanners advertise their presence on the LAN, and how clients discover them..

- The RESTful API chapter describes how clients send commands, and receive images and metadata from the scanner using HTTPS..

TWAIN Cloud

This document describes how to manage and enumerate scanners in the cloud, and how to communicate with them using the Client-Scanner API.

Certification

This document contains chapters describing how to test applications and scanners for compliance using TWAIN Direct, TWAIN Local and TWAIN Cloud.

PDF/raster

This document describes the format of a PDF/raster image file. Application writers may use existing PDF libraries to read these files, new code is not required.

Advice for Application Writers

Developers are advised to consider the following steps when adding TWAIN Direct to their application:

- Read this User's Guide
- Select either TWAIN Local and/or TWAIN Cloud support and read those documents
- Select a library for parsing JSON (refer to the References section for links)
- Select a library for parsing PDF/raster (refer to the References section for links)
- Review the Task and Metadata chapters to determine what the application will support
- Examine the TWAIN Direct sample code (refer to the References section for links)
- Perform Certification Tests

Advice for Scanner Vendors

Scanner vendors are advised to consider the following steps when adding TWAIN Direct to their devices:

- Read this User’s Guide
- Select either TWAIN Local and/or TWAIN Cloud support and read those documents
- Select a library for parsing JSON (refer to the References section for links)
- Select a library for generating PDF/raster (refer to the References section for links)
- Review the Task and Metadata chapters to determine what the scanner will support
- Examine the TWAIN Direct sample code (refer to the References section for links)
- Perform Certification Tests

Reasons for Creating TWAIN Direct

In today's environment, when a developer decides to add scanning capabilities to their application, they encounter the following issues:

- Image Capture APIs are generally limited to or best supported by the C and C++ programming languages. Third party toolkits are required when interfacing to other popular languages or web browsers.

- Developers must decide which Image Capture API to use (e.g., TWAIN, WIA, ISIS, SANE, ICA, etc). This decision is influenced by the operating system the application runs on, and the APIs supported by the scanner. This makes it harder for developers to switch to different scanner products, or support new ones.

- Image Capture APIs are complex, reflecting the native complexity of scanners, which support a large number of possible settings (e.g., compression, brightness, contrast, region-of-interest, deskew, barcode detection, etc). The TWAIN Specification currently describes more than 150 capabilities. In additional of the standard settings scanner vendors include their own custom capabilities. This complexity drives application developers to seek a "lowest common denominator" solution to their problems, even when the scanner is capable of producing a better user experience.

- Instead of focusing solely on solving the end user's problem, the bulk of application development time is devoted to defensive programming dealing with the variable feature set of scanners and the reliability of their drivers. Development of a robust scanning application takes weeks or months.

- Scanner vendors supply drivers for their devices (e.g., TWAIN, WIA, ISIS, SANE, ICA, etc). End users and IT personnel must install these drivers on their systems and upgrade them as needed. In this legacy model applications talk to drivers and drivers talk to scanners. The communication between a driver and a scanner is proprietary, making it harder for developers to diagnose problems without the help of the scanner vendor.

- The current Image Capture APIs were developed for peripheral communication protocols (e.g., IEEE 1284, SCSI, USB, etc). Application control of network scanners has not been standardized in a way that is adopted across the industry for all operating system platforms.

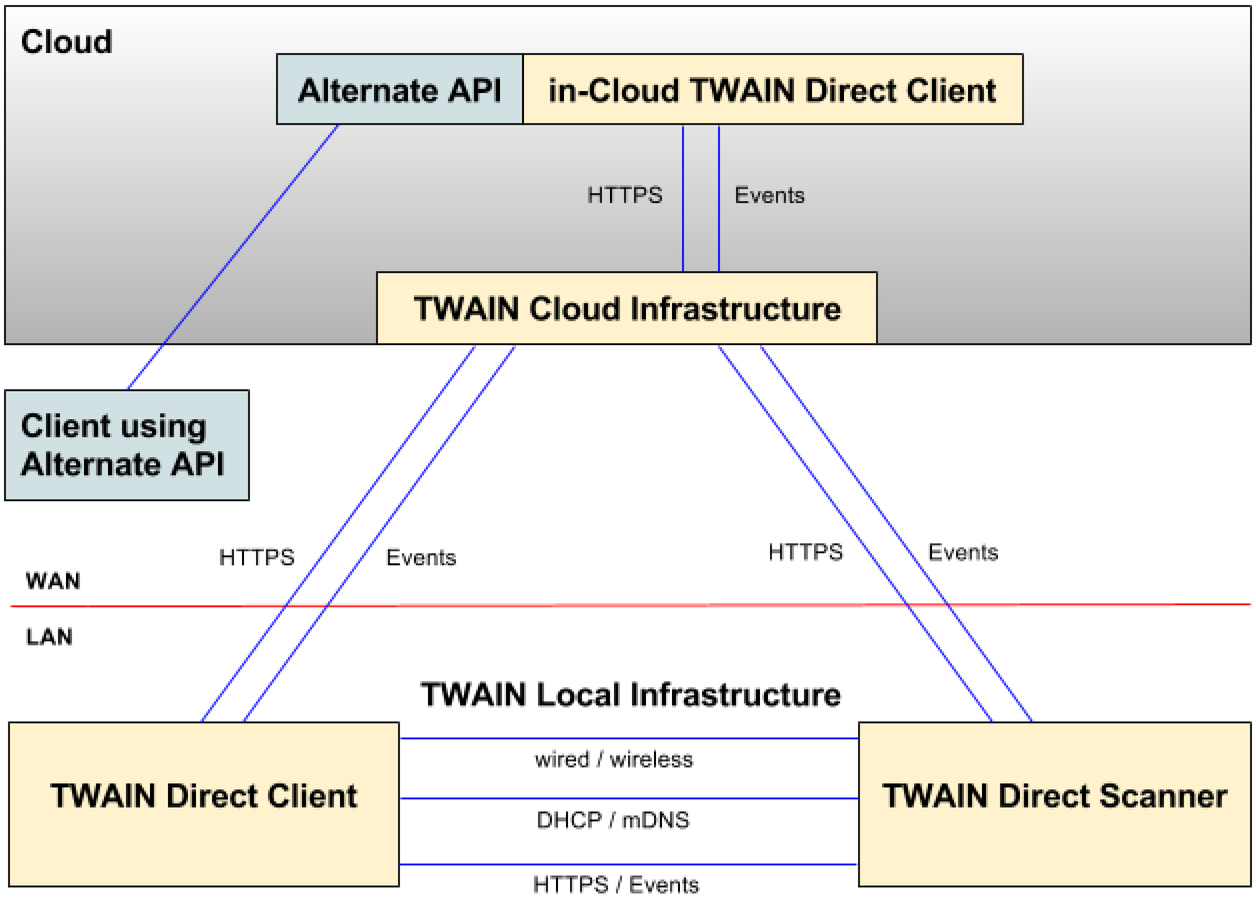
TWAIN Direct

What is TWAIN Direct?

- It is a protocol and a language that allows applications to talk directly to scanners without vendor specific drivers.

- It is an initiative to minimize the coding developers need to support fully featured image capture solutions in their applications.

The primary actors are the Client and the Scanner, which communicate through a local area network or the cloud. Clients can be in desktop or mobile devices, they can also be in the cloud, exposing alternate APIs, but using TWAIN Direct with the scanner. The protocol uses a mix of RESTful commands and events to minimize I/O and maximize performance.



TWAIN Local and TWAIN Cloud may be thought of “pipes” forming the infrastructure for finding and connecting with scanners. TWAIN Direct tasks, metadata, and PDF/raster images are the “water” flowing in the “pipes” and represent the actual communication between a client and a scanner.

Goals

- **Support direct communication between applications and scanners:** TWAIN Direct is not tied to any particular wired or wireless protocol, however, given the growing interest in mobile devices, the TWAIN Working Group is focusing TWAIN Direct on network connectivity. This network can be through a Cloud or direct communication between the user’s device and the scanner. This eliminates the need for vendor specific scanner drivers, reducing the support requirements for both scanner vendors and IT departments. Please refer to the documents on TWAIN Local and TWAIN Cloud for more information.

- **Simplify development:** Target the effort to the needs of application writers. Adding basic image capture support to an application should be measured in hours or days. The TWAIN Working Group supplies sample code and tasks in addition to the TWAIN Direct specification to help developers get started. TWAIN Direct uses HTTPS for its communication, JSON for its data, and PDF/raster for its images. It can be easily coded in any modern programming language, including scripting languages like python.

- **Support the full functionality of scanners:** Seamlessly integrate custom features from one or more vendors within the same TWAIN Direct language. Custom features are the driving force behind new additions to TWAIN Direct, so the standard makes it easy and safe to include them. An application can recommend specific scanners for the best user experience, while still working with less feature rich scanners. Over time custom features may be added to the specification, simplifying development for application writers.

- **Provide applications access to all of the metadata associated with an image and seamlessly integrate custom metadata without risk of namespace collisions:** The TWAIN Specification provides the starting point for standardized metadata items offered within TWAIN Direct.

- **Describe a baseline format for images that all scanners support, which can be easily expanded to encompass future functional requirements:** PDF/raster replaces older, less well defined image file formats. The format is verifiable as a valid subset of the existing PDF file format, but constrained in such a way that application writers can extract the image data without requiring a PDF library. The format is friendly to scanner vendors, who must stream image data without modifying it.

- **Maintain version independence:** All versions of TWAIN Direct are interoperable across all applications and scanners. The TWAIN Working Group backs this design concept with more than 25 years of API management experience with the classic TWAIN Specification.

- **Emphasize success:** Scanners must deliver images unless the application states otherwise. For example, if an application *requests* JPEG compression, but the scanner cannot deliver it, then the scanner delivers its default (ex: uncompressed rgb24 raster data). If the application *requires* that JPEG compression is used, and the scanner cannot deliver it, then and only then is the scanner required to return a configuration error.

- **Anticipate the future:** Commonly used Ether and Wi-Fi connections are not currently as fast as the best peripheral communication protocols (ex: USB 3.0). This will not always be the case. Faster scanners will eventually migrate to this platform, and TWAIN Direct is designed to be ready for when that happens.

Adoption

To facilitate adoption by application writers, and ease support by scanner vendors, TWAIN Direct proposes four different support paths:

- **Legacy Support**: The TWAIN Working Group provides open source code of a service that allows existing TWAIN scanners to be accessed using TWAIN Direct. This permits immediate adoption of the Specification by application writers. TWAIN Direct tasks can pass through the TWAIN v2.4 interface, which can also return TWAIN Direct metadata and PDF/raster images.

- **Virtual Scanner Support**: Vendors of existing network scanners can create a network component that exposes TWAIN Direct to applications, but communicates with the scanner using proprietary protocols.

- **Sidecar Support**: The TWAIN Working Group has demonstrated a way for existing USB scanners to expose a TWAIN Direct interface without modifying the base hardware of the scanner. The scanner is plugged into a device that implements TWAIN Direct. This device may come from the scanner vendor or a third party provider. This benefits IT departments by removing the need for legacy drivers on user's desktops.

- **Native TWAIN Direct Scanners**: Incentivize scanner vendors to natively support

TWAIN Direct in their scanners.

Versionless Interfaces

Any TWAIN Direct application must be able to communicate with any TWAIN Direct scanner, regardless of which version of the TWAIN Direct Specification was used for their implementation. Put another way, new versions of the TWAIN Direct specification never require applications or scanners to change their interfaces to communicate with existing products.

The only exception to this rule is for reasons of security. If some part of the TWAIN Direct specification must be modified in a way that breaks backwards compatibility, it will be clearly called out and advertised through the TWAIN Local and TWAIN Cloud interfaces.

TWAIN Direct uses the following rules to support a versionless interface:

- TWAIN Local and TWAIN Cloud protocols are focused on enumerating and connecting to scanners. The goal will always be to avoid changes to their design.

- The state machine for RESTful commands parallels the one used by Classic TWAIN for 25 years, its durability is proven.

- TWAIN Direct minimizes exposure to changeful standards. For instance, the TWAIN Local and TWAIN Cloud RESTful APIs do not reply upon the HTTP Status codes, except for simple pass/fail behavior (it’s either 200, or it isn’t).

- Key/value pairs are not used in the RESTful APIs.

- Use of the commandId in the RESTful API makes all commands idempotent (even ones using POST), reducing the chance of side effects in new or existing content.

- The TWAIN Direct language is where new features appear. Its design emphasizes success, unrecognized content is ignored by default.

- Existing content may be deprecated, but must never be obsoleted. The TWAIN Working Group may recommend a new feature over an older one, but once a feature is defined, it remains defined that way forever.

- In the unlikely event a new feature is made mandatory, its default value must match the behavior of prior versions of the TWAIN Direct specification.

TWAIN Direct Syntax

JSON

TWAIN Direct is formatted using JSON, a link pointing to more information about JSON is included in the References section.

JSON Schema

TWAIN Direct does not use or recommend the use of a JSON Schema in either applications or scanners. This decision was not lightly taken, but results from the following requirements:

- TWAIN Direct is versionless. Applications and scanners must be able to interoperate both now and years in the future. A schema naturally locks an API into a particular version.

- TWAIN Direct emphasizes success. If TWAIN Direct content contains properties or values unrecognized or unsupported by a scanner, the default action is to ignore them and continue processing the rest of the task.

- When the vendor property is specified at any level in a TWAIN Direct task, that content is fully defined by a scanner vendor, and not by the TWAIN Working Group. All that is required is that the content is valid JSON. All scanners must be able to skip over this content, when they detect that it doesn't apply to them.

A JSON Schema is an important convenience for application writers and scanner vendors, the TWAIN Working Group is addressing this by providing sample code that correctly parses TWAIN Direct tasks.

Note that this applies both to the protocol and the language. As TWAIN Direct evolves TWAIN Direct tasks and the RESTful API for TWAIN Local and TWAIN Cloud will be enhanced. It is a primary goal of the TWAIN Working Group to advance the specification in ways that do not require application writers or scanner vendors to change code in existing products.

Stylistic Conventions

TWAIN Direct is generally in agreement with the Google JSON Style Guide (a link is included in the References section). The salient points are as follows:

- Property names are selected for the clarity of their meaning.

- Property names are camelCased: the first word is lowercase, subsequent words are capitalized, this applies to abbreviations: (ex: meterCpu, not meterCPU).

- The first character of a property name is a lowercase alphabetic character in the range [a - z], subsequent characters must be alphanumeric (ex: theData, not TheData).

- Property names and values must be encased within double quotes [U+0022 "].

- The property names for JSON arrays are plural (ex: actions).

- JavaScript keywords are reserved and may not be used for property names, this applies to both the TWAIN Direct standard and vendor customizations. It's intended to make sure that TWAIN Direct tasks can be used with as many programming languages as possible. The same applies to words reserved by the Communication Managers. Scanner vendors need to pay especial attention to this when added custom content.

- The JSON integer type only supports signed 32-bit values, to send unsigned 32-bit or 64-bit values use the string type. The TWAIN Direct Specification calls out instances where this may happen.

- Base64 encodes content that would be invalid if transmitted as a JSON string. Properties that encode their data using Base64 must be prefixed with the name "base64", (ex: base64Data).

TWAIN Direct Actions

TWAIN Direct tasks contain zero or more actions. A task with no explicit action is a “null task”, and responds with an empty action result that includes a status. For example:

Task

```
{
}

-or-

{
  "actions": [
  ]
}

-or-

{
  "actions": [
    {
      "action": ""
    }
  ]
}
```

Response

```
{
  "actions": [
    {
      "action": "",
      "result": {
        "status": "success"
      }
    }
  ]
}
```

When actions are explicitly named, the scanner acts upon them in the order they appear in the task, they have immediate consequences, they return a result, and may return additional data. For example:

Task

```
{
  "actions": [
    {
      "action": "firstAction"
    },
    {
      "action": "secondAction"
    }
  ]
}
```

Response

```
{
  "actions": [
    {
      "action": "firstAction",
      "result": {
        "status": "success"
      }
    },
    {
      "action": "secondAction",
      "result": {
        "status": "success"
      }
    }
  ]
}
```

Standard TWAIN Direct actions and vendor actions can be mixed in the same task.

The description for a TWAIN Direct action indicates the consequence of calling that same action more than once within the same task.

TWAIN Direct Configuration Topologies

The “configure” action tells the scanner how to capture images from sheets of paper. It can be simple, or it can be complex, depending on the capabilities of the scanner and the needs of the application.

Scanners provide images from one or more physical image capture elements built into their packaging. These capture elements may or may not be independently configurable. The collection of configurable capture elements within a given scanner are referred to as its topology.

This section examines common topologies. Understanding these concepts helps to clarify the rationale behind the hierarchical format of a TWAIN Direct task.

Flatbed

This is the simplest scanner, and the one most often seen on low end multifunction devices. Paper is placed on a glass surface. The user requests one kind of image format: black-and-white, grayscale or color. The scanner produces one image for each scan.

Some flatbeds support image segmentation for photographs or business cards. The user places one or more items on the glass surface. The scanner returns one image for each item that it discovers on the glass.

Automatic Document Feeder (ADF)

One or more sheets of paper are loaded into a feeder or a hopper. The user requests one kind of image format: black-and-white, grayscale or color.

Simplex feeders capture an image for just the front side of the sheet of paper. Duplex scanners return an image for the front and the rear of the sheet of paper. Some duplex scanners permit scanning from the rear only, and may support independent image processing settings, like color for the front of a sheet and grayscale for the rear.

ADF with Flatbed

An ADF and a flatbed are paired together, with all the features described above. In addition the scanner may automatically scan from the flatbed if paper is not currently loaded into the ADF's feeder or hopper.

Production Scanners

Production scanners incorporate all the features described above. They may include automatic color detection and multi-pixel formats from a side of a sheet of paper.

With automatic color detection the scanner selects the best pixel format for the captured image data. An application developer may opt to specify independent image processing settings for color and grayscale, even though the scanner will only deliver one of the two pixel formats in the final image.

With multi-pixel format mode the scanner delivers more than one complete image for a side of a sheet of paper. For instance, capturing both a color image for archival purposes and a black-and-white image sent to an OCR engine to extract text.

Topology Breakdown

To cover the previous examples, and any future variations scanner vendors may develop, the TWAIN Direct defines the following breakdown for a scanner topology (note that not every action has to be a "configure" action):

```
action AND action AND action…
        stream OR stream OR stream…
                source AND source AND source…
                        pixelFormat OR pixelFormat OR pixelFormat...
                                attribute AND attribute AND attribute…
                                        value OR value OR value...
```

An action specifies a command to a scanner. Actions are performed in sequence from top to bottom in the task. All of the actions in a task are attempted. A "configure" action contains zero or more streams.

A stream contains a collection of sources. The first stream in an action that the scanner supports is used to capture images, all others are ignored. A stream contains zero or more sources.

A source designates a physical capture element, like the front and rear of a feeder, or a flatbed. All of the sources in a stream contribute to the flow of images returned from the scanner. A given source may appear more than once within the same stream, which provides multi-pixel format support. A source contains zero or more pixelFormats.

A pixelFormat combines a color space and a bit depth, such as rgb24. If more than one pixelFormat is specified within a single source, the scanner selects the best match for the captured image. A pixelFormat contains zero or more attributes.

An attribute is a configurable item, such as compression or resolution. All of the attributes under a pixelFormat are attempted by the scanner. An attribute contains zero or more values.

A value is applied to an attribute, such as jpeg for compression or 300 for resolution. The first value that the scanner can support is selected, all others are ignored.

Topology Combinations

An application creates a task's topology with the goal of capturing image data in the way that best suits its goals. The application decides on the complexity of the topology and what constitutes failure, that is, under what conditions the scanner must report that it cannot perform the task. The scanner examines the topology of the task and determines what it's able to support. The default behavior is to ignore problems in the task and return an image. TWAIN Direct tasks "emphasize success".

With this in mind the application writer can do the following:

- Specify more than one stream in an action. The most common use is to support capturing data from a flatbed if no feeder is available, or the feeder has no paper in it. It may also be used to switch to the next stream if an exception requests it on an unsupported setting. It is important to remember that only one stream is selected for a given action.

- Specify more than one source in a stream.   For devices that support it, this is the way to select independent settings for feederFront, feederRear and flatbed sources. It's also the way to create multi-pixelFormat output, such as capturing both color and grayscale images on one side of a sheet of paper, by specifying the same source more than once in a stream.

- Specify more than one pixelFormat in a source. In this case the scanner selects the best match for the image that it's captured. If the image contains enough color content, the scanner returns a color image, otherwise it returns grayscale or black-and-white.

- Most applications specify multiple attributes in a pixelFormat. As a general rule an application only sets attributes that it cares about, and relies on the scanner's default values for the rest.

- Specify more than one value for an attribute. The scanner selects the first supported value. This allows the application writer to offer specific values within an attribute ordered from most to least preferred.

TWAIN Direct Task

A TWAIN Direct task consists of one or more actions. A TWAIN Direct task without any actions is a null task. A device reports success when it receives a null task, but takes no other action.

A single task with multiple actions is functionally equivalent to several tasks run in order under the protection of a single session lock.

The scanner runs every action specified in a task, unless an exception prevents it.

JSON

This figure shows the basic structure of a TWAIN Direct task in JSON format, focusing on the topology, which implies that we're talking about a "configure" action:

```
{
  "actions": [
    {
      "action": "#the action#",
      "streams": [
        {
          "name": "#the name of an stream#",
          "sources": [
            {
              "source": "#the image source#",
              "pixelFormats": [
                {
                  "pixelFormat": "#the pixel format#",
                  "attributes": [
                    {
                      "attribute": "#the attribute#",
                      "values": [
                        { "value": "#a value#" }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Task Data Structure

The JSON format of a TWAIN Direct task maps to the following data structure. Refer to the appendix for examples of TWAIN Direct tasks that can be used to fill this structure.

```
// The TWAIN Direct task
mandatory structure task

  // The actions for the task, evaluated in order of appearance
  optional array of structure actions

    // The identification of the action (ex: configure or scan)
    optional string action

    // The exception for this action
    optional string exception = { default is “ignore” }

    // The vendor for this action
    optional string vendor = { default is TWAIN Direct }

    // The streams for this action, evaluated in order of appearance
    optional array of structure streams

      // The name of  the stream
      optional string name= { if a name isn’t provide the scanner fills in the name using the format “stream%02d” }

      // The exception for this stream
      optional string exception= { default is inherited from action’s exception, or “nextStream” or “ignore” }

      // The vendor for this stream
      optional string vendor = { default is inherited from action’s vendor }

      // The sources for this stream, evaluated in order of appearance
      optional array of structure sources

        // Source of images (ex: any, feeder, feederFront, flatBed, etc)
        optional string source = { default is “any” }

        // The exception for this source
        optional string exception = { default is inherited from stream’s exception }

        // The vendor for this source
        optional string vendor = { default is inherited from stream’s vendor }

        // The pixelformats for this source, evaluated in order of appearance
        optional array of structures pixelformat

          // Format of the image (ex: bw1, gray8, rgb24, etc)
          optional string pixelformat = { default set by device }

          // The exception for this pixelFormat
```

```
            optional string exception = { default is inherited from source's exception }

            // The vendor for this pixelFormat
            optional string vendor = { default is inherited from source's vendor }


            // The attributes for this pixelformat, in any order
            optional array of structure attributes

              // The identification of the attribute
              optional string attribute

              // The exception for this attribute
              optional string exception = { default is inherited from pixelFormat's exception }

              // The vendor for this attribute
              optional string vendor = { default is inherited from pixelFormat's vendor }

              // One or more values for this attribute, attempted in order of appearance
              optional array of structure values

                // The exception for this value
                optional string exception = { default is inherited from attribute's exception }

                // The vendor for this attribute
                optional string vendor = { default is inherited from attribute's vendor }

                // A single value
                optional string value

              end values
            end attributes
          end pixelformats
        end sources
      end streams
    end actions
  end task
```

Error Handling

There are three kinds of errors that a scanner can encounter when processing a task:

- JSON syntax errors
- TWAIN Direct topology errors
- Unsupported properties and values

A JSON syntax error prevents the scanner from processing the complete task. At a minimum it reports back the character offset of the error. Scanner vendors may opt to return more information about the error, if they have it available.

A TWAIN Direct topology error occurs when topology entries appear out of order. For example, the following task is missing the stream object, which must appear within the action object, and which contains the sources object. In this case the scanner reports that the topology is in error. At a minimum the scanner must provide a dotted key path to the error. In the following example it would be “actions[0].sources”.

```
{
  "actions": [
    {
      "action": "configure",
      "sources": [
        {
          "source": "feeder",
          "pixelFormats": [
            {
              "pixelFormat": "rgb24"
            }
          ]
        }
      ]
    }
  ]
}
```

Unsupported properties and values can occur anywhere within a task. The way they are handled is determined by the exception system, which is described further below in the section titled TWAIN Direct Task Reply and Error Reporting. At a minimum the scanner must provide a dotted key path to the error.

Power-On Defaults

Vendors determine the power-on default settings for their scanners, which includes the default topology, and all associated attributes. These defaults are always applied before processing any stream defined in an action. This way each stream can be trusted to build on the same initial settings for a given scanner model, without being affected by the settings of a previous stream.

To show this in action, consider a scanner that has both a feeder and a flatbed and only supports a pixelFormat of gray8. At power-on this scanner defaults to the flatbed.

This next task configures the scanner to use its power-on defaults, so the gray8 image comes from the flatbed.

```
{
  "actions": [
    {
      "action": "configure"
    }
  ]
}
```

This next task attempts to capture a color image from the feeder. The scanner cannot support this, so it moves to the next stream. When the second stream is entered the scanner resets to its power-on defaults. Note that the second stream does not specify a source. The end result is a gray8 image that comes from the flatbed.

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "source": "feeder",
              "pixelFormats": [
                {
                  "pixelFormat": "rgb24"
                }
              ]
            }
          ]
        },
        {
          "sources": [
            {
              "pixelFormats": [
                {
                  "pixelFormat": "gray8"
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

The TWAIN Direct Specification recommends power-on defaults for many attributes, but unless otherwise specified does not require specific values. This allows scanner vendors to support applications in the ways they deem most appropriate for their customers.

Vendor Customizations

Innovation occurs when scanner vendors add features to their products to better serve existing customers, or reach out to new market segments. Based on experience with TWAIN it's expected that the interfaces to these innovations are revealed to the TWAIN Working Group only after the new marketing opportunity has been fully productized.

It is only at this point the TWAIN Working Group develops a standard interface for the new feature. Depending on the number of scanner vendors supporting custom versions of the new feature, this means that application writers are confronted with multiple ways of accessing it. The TWAIN Direct language provides a way to easily integrate custom functionality into tasks at every point along their development.

TWAIN Direct supports the "vendor" property. This property is a string that is universally unique to a particular scanner vendor. TWAIN Direct defines its own string (as a UUID described in the References section), which is the default value for all tasks.

A task may include a "vendor" property anywhere within its topology, this has the following effect on the task:

- The string must be unique. The TWAIN Working Group does not maintain a registry for these names. Two recommended string types are: UUIDs and reverse-DNS. A UUID has the benefit of brevity and a high-degree of uniqueness. A reverse-DNS name has the benefit of readability. The reverse DNS name for TWAIN Direct is: org.twaindirect.www.

- All content within that specific JSON property and the properties contained within it are only recognized by the scanner vendor's products that match the vendor property. All other scanners must ignore this content.

- The only requirement for the vendor's content is that it must be valid JSON, following the style and restrictions for JSON outlined earlier within this Specification.

- If a scanner supports both the TWAIN Direct standard form of a command and a vendor specific form, and both appear within the same task in such a way that a choice must be made between the two, then the scanner must select the first one it encounters in the task. This gives application writers the ability to determine which form they prefer the scanner to use.

Sample Situation

To show this in action we'll demonstrate with a fictitious feature, and show how application writers benefit from TWAIN Direct's language.

Company Acme (com.companyacme.www) offers a new custom attribute called "fizzbin", which application writers want to use. A task supporting it takes the following form:

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "source": "any",
              "pixelFormats": [
                {
                  "pixelFormat": "rgb24",
                  "attributes": [
                    {
                      "attribute": "fizzbin",
                      "vendor": "com.companyacme.www",
                      "values": [
                        { "value": "kronk" }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Company Pinnacle (com.companypinnacle.www) also offers a new custom attribute also called "fizzbin" (because it's a natural name to select) with similar functionality, which application writers also want to use, but with a different name for the value. A task supporting it as well as Acme's version takes the following form:

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "source": "any",
              "pixelFormats": [
                {
                  "pixelFormat": "rgb24",
                  "attributes": [
                    {
                      "attribute": "fizzbin",
                      "vendor": "com.companyacme.www",
                      "values": [
                        { "value": "kronk" }
                      ]
                    },
                    {
                      "attribute": "fizzbin",
                      "vendor": "com.companypinnacle.www",
                      "values": [
                        { "value": "shralk" }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

If these vendors or other vendors propose the new feature to the TWAIN Working Group, then it may be added to a future version of the TWAIN Direct Specification. In this case we end up with the following task, which supports both the two legacy interfaces and the new standard interface (which in this example uses Company Acme's name for the value), with the standard form being preferred if a specific scanner is capable of handling both forms.

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "source": "any",
              "pixelFormats": [
                {
                  "pixelFormat": "rgb24",
                  "attributes": [
                    {
                      "attribute": "fizzbin",
                      "values": [
                        { "value": "kronk" }
                      ]
                    },
                    {
                      "attribute": "fizzbin",
                      "vendor": "com.companyacme.www",
                      "values": [
                        { "value": "kronk" }
                      ]
                    },
                    {
                      "attribute": "fizzbin",
                      "vendor": com.companypinnacle.www",
                      "values": [
                        { "value": "shralk" }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Exceptions

TWAIN Direct's reduces the amount of code that an application needs to control a scanner. An important way of doing this is through exceptions. The exception system allows an application to present choices to the scanner, which are used when the scanner is unable to configure itself in exactly the way the application wants.

Vendor marked items unrecognized by the scanner must be ignored For instance, if one of the streams in a task is tagged with a "vendor" string, and that string does not match the current scanner or the TWAIN Direct string, then that stream and all of its contents are skipped over and ignored, regardless of any exception settings.

The exception system offers the following options for each part of the topology.

- **action:** ignore, fail, nextAction
  - **stream:** ignore, fail, nextAction, nextStream
    - **source:** ignore, fail, nextAction, nextStream
      - **pixelFormat:** ignore, fail, nextAction, nextStream
        - **attribute:** ignore, fail, nextAction, nextStream
          - **value:** ignore, fail, nextAction, nextStream, (and implied "next value")

Exceptions are inherited and can be overridden at any point within the topology. For example, if **fail** is selected in the source it becomes the default for all pixelFormats, attributes and values under it, unless one of them specifies their own exception to override it.

The default exceptions are laid out as follows. To make this as clear as possible, we'll show one action with one stream, one action with multiple streams, and multiple actions with multiple streams.

In this instance all parts of the task ignore problems with any specific property, and proceed to the next property at that same level. If is also valid to think of it as a generic "next". For instance, if there are three attributes in a pixelFormat, and the second one can't be honored by the scanner, it's ignored (by default) and continues on to the third attribute.

<u>One action with one stream</u>

- **action:** exception: **ignore**
  - **stream:** exception: **ignore**
    - **source:** exception for all sources: **ignore**
      - **pixelFormat:** exception for all pixelFormats: **ignore**
        - **attribute:** exception for all attributes: **ignore**
          - **value:** exception for last value: **ignore**

In this instance, a problem with any element within the first stream causes the scanner to discard and proceed to the second stream. Problems with the last stream are ignored.

<u>One action with multiple streams</u>
action: exception: **ignore**

stream:
exception for all streams except the last: **nextStream**
source: exception for all sources: **nextStream**
pixelFormat: exception for all pixelFormats: **nextStream**
attribute: exception for all attributes: **nextStream**
value: exception for last value: **nextStream**

… (zero or more additional streams) …

exception for the last stream: **ignore**
source: exception for all sources: **ignore**
pixelFormat: exception for all pixelFormats: **ignore**
attribute: exception for all attributes: **ignore**
value: exception for last value: **ignore**

In the final example we show how the scanner defaults the exceptions in the case of multiple actions and multiple streams. The only difference from the previous example is that the last stream in any given action, except for the last action, defaults to **nextAction**.

<u>Multiple actions with multiple streams</u>
actions:
exception for all actions except the last: **nextAction**

stream:
exception for all streams except the last: **nextStream**
source: exception for all sources: **nextStream**
pixelFormat: exception for all pixelFormats: **nextStream**
attribute: exception for all attributes: **nextStream**
value: exception for last value: **nextStream**

… (zero or more additional streams) …

exception for the last stream (use this if it's the only stream): **nextAction**

```
        source: exception for all sources: nextAction
            pixelFormat: exception for all pixelFormats: nextAction
                attribute: exception for all attributes: nextAction
                    value: exception for last value: nextAction

… (zero or more additional actions) …

exception for the last action (use this if it’s the only action): ignore

stream:
    exception for all streams except the last: nextStream
        source: exception for all sources: nextStream
            pixelFormat: exception for all pixelFormats: nextStream
                attribute: exception for all attributes: nextStream
                    value: exception for last value: nextStream

    … (zero or more additional streams) …

    exception for the last stream (use this if it’s the only stream): ignore
        source: exception for all sources: ignore
            pixelFormat: exception for all pixelFormats: ignore
                attribute: exception for all attributes: ignore
                    value: exception for last value: ignore
```

Exception: ignore

This is the default exception for TWAIN Direct tasks. The scanner ignores any configuration problems it encounters.

- The explicit use of “ignore” by any stream in a task means that the scanner uses that stream if it reaches it, any streams that follow are ignored (so be careful).

- The use of “ignore” by any source means that source will be included in the stream, if the scanner can support multiple sources. In this instance the scanner will include the sources it can and ignore the others. If a scanner is able to recognize more than one source, but can only support one of them at a time, it must select the first one it’s able to support.

- The use of “ignore” by any pixelFormat means the scanner will consider it as a candidate for the stream, if the stream supports automatic pixelFormat detection. If a scanner is able to recognize more than one pixelFormat, but can only support one of them at a time, it must select the first one it’s able to support.

- Attributes that are not recognized are ignored.

- Values for attributes that are not recognized or cannot be set are ignored and the attribute retains its power-on default value.

Exception: fail

If during the course of processing a task the scanner runs into an item that it cannot recognize or support, and that item either inherited the “fail” exception from an outer portion of the topology or has it explicitly requested within the same item, then processing of the task stops and a failure is returned.

Application writers must be sparing in the use of the “fail” exception, reserving it for situations where returning an image that doesn’t exactly match the needs of the user is worse than returning no image at all.

Exception: nextAction

When there are multiple actions in a task, “nextAction” becomes the default exception for all except the last action (or the only action), which defaults to “ignore”.

If “nextAction” is specified, then an inability to recognize a property or to set a value causes processing of the current action to stop, and the scanner proceeds to the next action.

If “nextAction” is specified for the last stream (or the only action) in an action, then an inability to recognize a property or set a value causes the processing of the current action to stop, and the scanner reports that the task failed. In other words, don’t do this.

Exception: nextStream

When there are multiple streams under an action, “nextStream” becomes the default exception for all except the last stream (or the only stream), which defaults to “ignore”.

If “nextStream” is specified, then an inability to recognize a property or to set a value causes processing of the current stream to stop, and the scanner proceeds to the next stream.

If “nextStream” is specified for the last stream (or the only stream) in an action, then an inability to recognize a property or set a value causes the processing of the current stream to stop, and the scanner reports that the task failed. In other words, don’t do this.

Exception: “next value”

This only applies to the values of an attribute and is always implied, there is no explicit way to set it. The default behavior is to try each value in sequence. If none of the values can be used, then the exception for that attribute is applied.

In this example the application requests the scanner to set itself to a resolution of 50. If that’s not available a value of 75 is requested. If neither value can be set then the attribute and therefore the entire task is set to fail.

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "source": "any",
              "pixelFormats": [
                {
                  "pixelFormat": "rgb24",
                  "attributes": [
                    {
                      "attribute": "resolution",
                      "exception": "fail",
                      "values": [
                        { "value": 50 },
                        { "value": 75 }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

TWAIN Direct Task Response and Error Reporting

Errors are handled by both the protocol and the language. Errors in the protocol, such as communication problems, state violations, or mistakes in the construction of the JSON, are reported by TWAIN Local and TWAIN Cloud. All other errors are handled by the TWAIN Direct language, and are described in this document.

Examples of errors in the TWAIN Direct language are provided in the Certification document for TWAIN Direct.

When successful the scanner reflects the portion of the task that it used. For example if a scanner with a feeder is given the following task with a "*configure*" action:

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "exception": "fail",
              "source": "feeder"
            }
          ]
        }
      ]
    }
  ]
}
```

The scanner responds with a success status and feedback about the task indicating that the "*feeder*" was used. Applications must not assume that the *action*, *results,* and *streams* properties occur in any special order, but scanner vendors should use this sequence:

```
{
  "actions": [
    {
      "action": "configure",
      "results": {
        "success": true
      },
      "streams": [
        {
          "sources": [
            {
              "source": "feeder"
            }
        ]
      }
    ]
  }
 ]
}
```

For the same task a flatbed responds with an error code and a key indicating where the problem occurred in the original task sent to the scanner:

```
{
  "actions": [
    {
      "action": "configure",
      "results": {
        "success": false,
        "code": "invalidValue",
        "jsonKey:actions[0].streams[0].sources[0].source"
      }
    }
  ]
}
```

The key is the dotted path to the first property with an unresolvable problem. In the above example the path is the source property; in the 0th element of the sources array; in the 0th element of the streams array; in the 0th element of the actions array. "*feeder*" isn't supported by a flatbed only scanner, and the task specified an exception of "*fail*", so the scanner aborted the task.

A more complicated example uses two streams, the first stream asks for images to be captured from the feeder, the second asks for images to be captured from the flatbed:

```
{
  "actions": [
    {
      "action": "configure",
      "streams": [
        {
          "sources": [
            {
              "source": "feeder"
            }
          ]
        },
        {
          "sources": [
            {
              "source": "flatbed"
            }
          ]
        }
      ]
    }
  ]
}
```

If the scanner has a feeder and paper is available to be scanned, the scanner’s response comes from the first stream:

```
{
  "actions": [
    {
      "action": "configure",
      "response": {
        "success": true
      },
      "streams": [
        {
          "sources": [
            {
              "source": "feeder"
            }
          ]
        }
      ]
    }
  ]
}
```

If the scanner is unable to support the feeder source (because one isn’t available or because paper isn’t present in the feeder), the response takes the form:

```
{
  "actions": [
    {
      "action": "configure",
      "response": {
        "success": true
      },
      "streams": [
        {
          "sources": [
            {
              "source": "flatbed"
            }
          ]
        }
      ]
    }
  ]
}
```

If the scanner is unable to support either source, perhaps because it has a planetary source or is getting images from pre-existing storage, then its response is:

```
{
  "actions": [
    {
      "action": "configure",
      "response": {
        "success": true
      }
    }
  ]
}
```

The lack of a specified stream indicates that the scanner's power-on defaults will be used.

Error: "invalidTask"

If there is an error in the construction of the task, for instance if the topology for the "*configure*" action is invalid, the scanner responds with "*invalidTask*" and a key pointing to the object in error. An empty key indicates that the problem is in the root object. For instance, the following task specified "*sources*" without placing it in a "*stream*"

```
{
  "actions": [
    {
      "action": "configure",
      "sources": [
        {
          "source": "feeder"
        }
      ]
    }
  ]
}
```

The scanner responds with an error, and indicates through the "*key*" that the problem was detected inside of the first action:

```
{
  "actions": [
    {
      "action": "",
      "response": {
        "success": false,
        "code": "invalidTask",
        "jsonKey": "actions[0]"
      }
    }
  ]
}
```

Error: “invalidJson”

Errors in the construction of the JSON string are necessarily detected by the protocol, and not the language. Please refer to the TWAIN Local and TWAIN Cloud documents for how these can occur and the way they are reported.

The Certification document for TWAIN Direct has a section where JSON errors are generated in the TWAIN Direct task.

PDF/raster Images

TWAIN Direct scanners produce finished images, that is, the image output from a scanner, when written to a file with the appropriate extension, is viewable by a program that recognizes that image file format.

TWAIN Direct scanners must default to outputting image data using the following format:

PDF/raster (Portable Document Format)

PDF/raster is designed as a wrapper for 24-bit color and 8-bit grayscale as JPEG or uncompressed raster data, and 1-bit packed black-and-white as Group4 or uncompressed raster data.

The format is a valid PDF, but the use of a library is optional. The format is simple and consistent enough to allow a developer to open it and extract the image data.

Links to PDF/raster information may be found in the References section.

PDF/raster images generated by TWAIN Direct scanners must include metadata, and be digitally signed. Refer to the TWAIN Direct - Metadata specification for more information.

Metadata

Image metadata comes in six broad categories:

- The status of the image. A success of true indicates that the image was successfully captured, false indicates a problem occurred.

- Data describing the image, including the width, the height, the image format and the byte size of the image.

- Data describing the relationship of the image to other content, including the source of the image (feederFront, flatbed, etc) and the ordinal number of the sheet that supplied the image. This information can be used to reconstruct the organization of the sheets of paper captured by the scanner.

- Data gleaned from the image, including barcode, MICR and patch codes.

- Data that accompanies an image, such as text strings printed on the document by the scanner (if printed after scanning takes place these string will not appear on the image).

- Vendor specific content, allowing scanner vendors to report more information than is described by this specification.

The data is organized as JSON, with objects grouping related properties. A typical example is shown below:

```
{
  "metadata": {
    "status": {
      "success": true
    },
    "address": {
      "imageNumber": 1,
      "imagePart": 1,
      "moreParts": "lastPartInFile",
      "sheetNumber": 1,
      "source": "feederFront",
      "streamName": "stream0",
      "sourceName": "source0",
      "pixelFormatName": "pixelFormat0",
    },
    "image": {
      "compression": "none",
      "pixelFormat": "bw1",
      "pixelHeight": 1650,
      "pixelOffsetX": 0,
      "pixelOffsetY": 0,
      "pixelWidth": 1280,
      "resolution": 150
    }
  }
}
```