

Maxwell[®] Application Programming Interface Guide

Table Of Contents

Maxwell Application Programming Interface Guide

Legal Statements.....	2
Restricted Rights Legend.....	2
Additional Resources.....	2
Introduction.....	1
Connecting and Command Syntax.....	2
Affect of Impairment Modes on Plugins.....	3
Shell Commands.....	3
Command Categories.....	4
Flow Match Control.....	4
Impairment Control.....	7
Plugin Control.....	10
Statistics.....	12
Log File Control.....	14
Miscellaneous Commands.....	14
BNF Summary for the Standard Impairment Server CLI.....	16
The Perl API.....	19
The Java API.....	19
The Python API.....	20

Legal Statements

© 2008-2010 by InterWorking Labs, Inc. All rights reserved. No part of this document may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language, in any form by any means, without the written permission of InterWorking Labs, Inc.

This system contains elements subject to following patent:

US Pat. 7,310,316.

Maxwell is a trademark of **InterWorking Labs, Inc.**

All other trademarks belong to their respective owners.

Restricted Rights Legend

This software is provided with RESTRICTED RIGHTS.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs(c)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable.

The "Manufacturer" for purposes of these regulations is InterWorking Labs, PO Box 66190, Scotts Valley, California 95067 U.S.A.

Additional Resources

The file system directory `/usr/local/iwl/docs/` on your Maxwell® Network Emulator machine contains this and other documentation. This document assumes you have read the following documents:

Maxwell® Installation and Quickstart Guide

Maxwell® Network Emulator and Impairment System Guide

Depending on the version of Maxwell you purchased, other documentation includes:

Maxwell® Plugin Programmer's Guide

Maxwell® TCP/IP Plugin Programmer's Guide

Maxwell® Automated TCP/IP Tests

Maxwell® SIP Tests

Maxwell® G.1050 Tests

To obtain additional support you may email InterWorking Labs at the e-mail address maxwell-support@iwl.com

Introduction

The Maxwell® Standard Impairment Server (the stdiserver process) can perform many types of impairments (such as dropping, duplicating, and delaying packets) on different packet flows. You may define multiple flows for each of the two data interfaces. This document describes how you can connect to the Standard Impairment Server and the syntax and semantics of the commands you may send to Standard Impairment Server to control its behavior.

Any programming language that is capable of opening a TCP/IP connection can be used to control Maxwell. Maxwell is supplied with a Java jar file, Perl Module, Python module, and shell commands that should allow you to quickly write programs and scripts to control Maxwell. (The Maxwell GUI itself uses the included Python module.) If you wish to use a language we have not included, we believe it should not be too hard to write appropriate command/response code.

The TCP/IP socket is called the *Remote API* and is used by internal and external TCP/IP client processes. The client processes connect to a *Command Line Server* inside the stdiserver process at a well-known TCP/IP port (default 7021). Once a connection has been established then the client may issue commands using the *Standard Impairment Server Command Line Interface*. All the programming language APIs are built upon the Remote API.

1. Remote API and shell commands are described in this document. Even if you only intend to use the Perl, Java, or Python APIs we have supplied, you should familiarize yourself with the Remote API commands.
2. Python API documentation is described in this document, but further details are in the servercomm.py and usage example files.
3. Perl API is introduced in this document, but further details are in the StdiServer.pm and usage example files.
4. Java API is introduced in this document, but further class details can be found here:
`/usr/local/iwl/stdijava_api/javadoc/index.html`.

Connecting and Command Syntax

In order for an external system to define flows and impairments, it must open a TCP/IP client connection to port 7021 on the Maxwell platform. Once a connection has been established to the server, the client may issue commands using the command line interface language. A set of shell commands exist on Maxwell itself which allows users to control it by logging into Maxwell via ssh and controlling it via shell scripts. These shell commands are almost syntactically and semantically identical to the set of commands in the command line interface command.

The command line interface language is a line-oriented text-encoded protocol. The commands are defined as text messages that are sent to the server, which sends text message responses back to the client. The BNF construction rules for the commands and responses are summarized below.

One or more white space characters must separate tokens from each other. White space characters are the space, tab, and carriage return characters. The newline character is used to terminate the commands and responses – *but not if* the newline appears in a quoted string or is preceded by a backslash (\) character. To create tokens containing white space, they must be enclosed by two double-quote (") characters. Note that command tokens are generally case insensitive. But in general you should assume command tokens and their arguments are case sensitive. Note also that if a newline is preceded by a backslash (but not inside a quoted string) then the backslash is to be treated as non-line terminated white space (that is, the command or response is continued on another line.)

Below are some examples of typical dialog. Lines longer than the width of the page have been split into multiple lines for presentation purposes, though the newline character only occurs at the end. The "CLIENT:" label precedes the command issued by the client and the server output is preceded by "SERVER:" label. Here's a simple example:

```
CLIENT: getnumflows
SERVER: Okay 4
```

This example's response indicates that up to 4 flow signatures may be defined. (These values can be set to larger values when the server is started.) Here are a few more examples of what typical command/response dialogs looks like:

```
CLIENT: setmatch 0 0 ipv4 addr src 199.184.129.0 255.255.255.0
SERVER: Okay
CLIENT: setimpair 0 0 drop 12.5 jitter 50.0 gaussian 200 30
SERVER: Okay
CLIENT: setmatch 0 1 ipv4 port dst 80 0xFF
SERVER: Okay
CLIENT: setimpair 0 1 delay 60
SERVER: Okay
CLIENT: setmatch 1 0 all
SERVER: Okay
CLIENT: setimpair 1 0 dup 1.5 couple 35.0
SERVER: Okay
```

Affect of Impairment Modes on Plugins

A frequently asked question our support gets is “Why aren't my setplugin commands working!?” What is typically happening is that the Standard Impairment Server is in the wrong **impairment mode** for that flow interface. There are three impairment modes for a flow interface: **forward all**, **drop all**, and **standard**. These modes affect processing as follows:

- Forward all: All matching packets are forwarded to the opposite interface for immediate transmission, bypassing all impairments, including plugin impairments! This is the “default” mode that flow interfaces are started with.
- Drop all: All matching packets are simply dropped, again bypassing all impairments, including plugin impairments.
- Standard: All matching packets are forwarded to the impairment “waterfall” for processing. If not dropped by one of the impairments, the packets eventually get processed by the plugin, if one is installed. Standard mode is entered when drop, duplicate, delay, rate, jitter, or the “null” impairments are specified.

To enter standard mode so your plugin “works,” you should enter the “null” impairment using the following command in addition to the setplugin command needed to activate the plugin:

```
setimpair flow interface
```

Note that entering drop, duplicate, delay, rate, or jitter impairments will also automatically cause that flow interface to enter standard mode. Just keep in mind that the setplugin command does not change the impairment mode, which is why the either the “null” impairment command above must be used or one that sets one drop, duplicate, etc.

Shell Commands

There are nineteen remote API and shell commands, grouped into six categories: flow match control, impairment control, plugin control, statistics, log file control, and miscellaneous. The commands are explained in detail below. The syntax for the shell commands is almost identical – but in the shell, the command name must be entered all lower case, whereas the remote API socket will accept mixed case. The shell commands also accept two command line “dash” arguments that the remote API does not: “-t *interval*” and “-p *port*”. If the “-t” option is provided, the shell command does not exit but repeatedly sleeps for *interval* seconds and reissues the command. If the “-p” option is provided, the shell command will attempt to connect to the provided local host *port* rather than default port 7021. Port redirect for the shell commands may also be performed by setting the shell variable REMOTEAPI_PORT.

Command Categories

Flow Match Control

- **getmatch** *Flow Interface*
Returns the current match criteria for the given interface flow. The syntax of the match criteria is the same as the syntax as that described in the **setmatch** command.
- **setmatch** *Flow Interface [zerostats] Match*
Sets the match criteria for the given interface flow. If zerostats is included, the statistic counts are set to zero. The syntax of match criteria is described below by the *Match* BNF element. In several places value/mask pairs are used, which have the following general rule: where the mask has a one bit, then the corresponding bit in the value must match the packet's bit for the packet to pass the match criteria. A zero bit in the mask means the corresponding bit in the packet can be any value.

```
Match :=          "none" | "all" | ["allbut"] ["bytes" Bytes] ["lan" Lan]
                ["ipv4" IPv4 | "ipv6" IPv6]
                ["trigger" {"basic" | "tcpudp"} uint32 uint32 uint32]

Lan :=           {"usesvlan" | "vlanid" uint16 Uint16Mask | "priority" uint8
                Uint8Mask | "type" uint16 Uint16Mask | "addr" {"src" | "dst" |
                "both" | "notused"} MacAddress MacAddressMask} [Lan]

IPv4 :=         {"port" {"src" | "dst" | "both" | "notused"} uint16 Uint16Mask |
                "addr" {"src" | "dst" | "both" | "notused"} IPv4Address
                IPv4AddressMask | "qos" uint8 Uint8Mask | "flags" uint8
                Uint8Mask | "proto" uint8 Uint8Mask} [IPv4]

IPv6 :=         {"port" {"src" | "dst" | "both" | "notused" } uint16 Uint16Mask
                | "addr" {"src" | "dst" | "both" | "notused" } IPv6Address
                IPv6AddressMask | "class" uint8 Uint8Mask | "flow" uint32
                Uint32Mask | "proto" uint8 Uint8Mask } [IPv6]

Bytes :=        OffsetBase Offset uint32 Uint32Mask [Bytes]
```

The semantics of each of the above elements are:

none:	No packets match; they are all forwarded to the next flow.
all:	All packets match.
allbut:	All packets are considered to match <i>except</i> those with the given byte, LAN, or IP settings.
bytes:	An array of one or more byte patterns follows this token.
<i>OffsetBase</i> :	One of the following string constants:
frame:	The matching should begin <i>Offset</i> bytes after the beginning of the MAC frame.
framedata:	The matching should begin <i>Offset</i> bytes after the end of the MAC frame header.

`ipdata:` The matching should begin *Offset* bytes after the end of the IPv4 or IPv6 header.

`tcpudpdata:` The matching should begin *Offset* bytes after the end of the TCP or UDP header.

Offset: The number of bytes after the *OffsetBase* to apply the match.

uint32: The value to match, subject to the *Uint32Mask*.

Uint32Mask: The match mask – each zero bit means "match all" otherwise match the value above.

`lan:` All the Lan header values (if specified) that follow must match:

`usesvlan:`

The MAC header must have a VLAN field of any value.

`vlanid uint16 Uint16Mask:`

The MAC header must have a VLAN field with the value *uint16* as masked by *Uint16Mask*.

`priority uint8 Uint8Mask:`

The MAC priority field must have the value *uint8* as masked by *Uint8Mask*.

`type uint16 Uint16Mask:`

The MAC type field must have the value *uint16* as masked by *Uint16Mask*.

`addr {src | dst | both | notused } MacAddress MacAddressMask:`

The MAC source, destination, or either address field must have the value *MacAddress* as masked by *MacAddressMask*.

`ipv4:` All the IPv4 header values (if specified) that follow must match:

`port {src | dst | both | notused } uint16 Uint16Mask:`

The IPv4 source, destination, or either port field must have the value *uint16* as masked by *Uint16Mask*.

`addr {src | dst | both | notused } IPv4Address Ipv4AddressMask:`

The IPv4 source, destination, or either address field must have the value *IPv4Address* as masked by *IPv4AddressMask*.

`qos uint8 Uint8Mask:`

The IPv4 QOS field must have the value *uint8* as masked by *Uint8Mask*.

`flags uint8 Uint8Mask:`

The IPv4 Flags field must have the value *uint8* as masked by *Uint8Mask*.

proto uint8 Uint8Mask:
The protocol field must have the value *uint8* as masked by *Uint8Mask*.

ipv6: All the IPv6 header values (if specified) that follow must match:

port {src | dst | both | notused } uint16 Uint16Mask:
The IPv6 source, destination, or either port field must have the value *uint16* as masked by *Uint16Mask*.

addr {src | dst | both | notused } IPv6Address IPv6AddressMask:
The IPv6 source, destination, or either address field must have the value *IPv6Address* as masked by *IPv6AddressMask*.

class uint8 Uint8Mask:
The IPv6 Class field must have the value *uint8* as masked by *Uint8Mask*.

flow uint32 Uint32Mask:
The IPv6 Flow field must have the value *uint32* as masked by *Uint32Mask*. Note that only the lower 24 bits are tested.

proto uint8 Uint8Mask:
The protocol field must have the value *uint8* as masked by *Uint8Mask*

trigger: Trigger delayed impairment matching is enabled and impairments begin, end, and repeat according to the last three integers.

basic | tcpudp
Select *basic* if only a single trigger state is valid at any one time or any type of packet must be selected in the other flow criteria. Select *tcpudp* when you wish to trigger on multiple matching TCP and UDP packets. See the System Guide for additional details.

uint32
Packets are impaired after this many packets have been matched after the triggering packet. If this value is zero, then the trigger packet is the first packet to be impaired.

uint32
Once impairments have begun, this indicates how many matching packets in a row should be impaired. This value should be set to zero to indicate an infinite number of packets.

uint32
This indicates how many times to repeat the trigger. Zero indicates infinite repetitions.

Impairment Control

- **getimpair** *Flow Interface*
Returns the current impairments being applied to the given interface flow. The syntax of the impairments is the same as that described in the **setimpair** command.
- **setimpair** *Flow Interface [zerostats] Impairment*
Sets the impairments to be applied to all packets that met the match criteria for the given interface flow. If zerostats is included, the statistic counts are set to zero. The syntax of impairments is described below by the *Impairment* BNF element.

```
Impairment := "forwardall" | "dropall" | [Drop] [Dup] [Delay] [Rate] [Jitter]
             [Corrupt]

Drop := "drop" percent ["couple" percent CouplingInterval ["slew"]]

Dup := "dup" percent ["couple" percent]

Delay := "delay" uint32

Rate := "rate" XmitRate MinPayload MaxPayload Overhead MaxQLength
        ActualRate ActualDuration

Jitter := "jitter" percent JitterTime ["reorderok"] ["couple" percent
        CouplingInterval ["slew"]]

Corrupt := "corrupt" ["payload"] percent

JitterTime := "fixed" uint32 | "linear" LowerBound UpperBound | "gaussian"
              Mean StdDeviation | "piecewise" Pieces

Pieces := "End" | Piece Pieces

Piece := PieceWidth {"fixed" uint32 | "linear" LowerBound UpperBound |
                    "gaussian" Mean StdDeviation}
```

The semantics of each of the above elements are:

forwardall: Send all packets immediately to the plugin, if one is loaded, or the outgoing interface. Puts the flow into the *Forward All* mode.

dropall: All matching packets are discarded. Puts the flow into the *Drop All* mode.

drop percent: Randomly select *percent* of the packets and discard them.

couple percent CouplingInterval [slew]:

When a packet is dropped, there is a *percent* chance that another packet arriving within *CouplingInterval* microseconds will also be dropped. If *slew* is included, the probability of a follow-on drop declines from *percent* to zero over the *CouplingInterval*.

dup percent: Randomly select *percent* of the packets and duplicate them.

`couple percent`: When a packet is duplicated, there is a *percent* chance that yet another copy is emitted . Increasing the coupling percentage increases the probability of triplicates, quadruplicates, and so on.

`delay uint32`: Delay each packet by *uint32* microseconds.

`rate XmitRate MinPayload MaxPayload Overhead QLength ActualRate ActualDuration`

The rate parameters control the transmission so that, on average, no more than *XmitRate* bits/second are transmitted out the opposite interface (a rate of zero disables rate impairment). When an Ethernet frame is received, only the bits in its data section are included in the rate computation – the frame preamble, header, FCS, and so on, are ignored. To accurately compute the packet delays, the number of overhead bits in the emulated outgoing link layer need to be included in the rate computation. To calculate the number of such overhead bits, the following aspects of the emulated link layer must be provided: the minimum and maximum payload bit size (*MinPayload* and *MaxPayload*, respectively) and the average number of bits in the emulated link layer's headers and and tails, if any (in *Overhead*.) To emulate discards properly, the maximum emulated transmission queue bit length must be provided in *QLength*. Lastly, for highest fidelity the actual measured transmission rate between Maxwell and the final endpoint is entered as *ActualRate* bits/second and actual measured transit delay is entered as *ActualDuration* microseconds. Note that setting *ActualRate* to zero excludes both *ActualRate* and *ActualDuration* from the rate computations.

`jitter percent JitterTime [reorderok]`:

Randomly select *percent* of the packets and delay them by a random *JitterTime* amount. If *reorderok* is included then non-delayed packets are allowed to "pass" delayed packets.

`couple percent CouplingInterval [slew]`:

When a packet is jittered, there is a *percent* chance that another packet arriving within *CouplingInterval* microseconds will also be delayed the same amount. If *slew* is included, the probability of a follow-on delay declines from *percent* to zero over the *CouplingInterval*.

`fixed uint32`: Delay selected packet by *uint32* microseconds.

`linear LowerBound UpperBound`:

Delay selected packet a random number of microseconds between *LowerBound* and *UpperBound*. All values between these bounds are equally probable.

`gaussian Mean StdDeviation`:

Delay selected packet a random number of microseconds centered on *Mean* with a probability that decreases according to a Gaussian distribution and a standard deviation of *StdDeviation* microseconds.

`piecewise:` Delay selected packet by a random number of microseconds where the probability function is composed of one or more pieces. Each piece may delay a packet using a different functional form: fixed, linear, or Gaussian.

PieceWidth: The probability of a function piece being selected is proportional to the *PieceWidth* value divided by the sum of all the *PieceWidth* values for this piecewise delay.

`corrupt [payload] percent :`

Randomly select *percent* of the bits in packets and invert them, this corrupting them. If “payload” is included then only the bits in the Ethernet payload are subject to random selection and corruption. The percent value is limited to a maximum of 0.3%. Up to 16 bits in any one packet may be corrupted.

Plugin Control

- `getpluginmeta`

This command retrieves information that describes the fields that are available to control the currently loaded plugin, if any. A plugin may be controlled with numeric input fields, checkbox fields, and radio button controls. The syntax of the response is described by the *PluginMeta* BNF element:

```
PluginMeta := "path" QuotedString "opts" QuotedString "ver" QuotedString "id"
              QuotedString "desc" QuotedString [CheckBoxes] [RadioButtons]
              [NumberFields]

CheckBoxes := "checkbox" Index Label Tooltip EnableExpression InitValue
              [CheckBoxes]

RadioButtons := "radiobuttons" Label Tooltip EnableExpression InitValue
               ButtonLabels

ButtonLabels := "end" | Index Label ButtonLabels

NumberFields := "number" Index Label Tooltip EnableExpression FieldType HardMin
                HardMax InitMin InitMax InitValue [NumberFields]
```

The semantics of each of the above elements:

`path` *QuotedString*: The full path name of the loaded plugin dynamic library.

`opts` *QuotedString*: The options string supplied to the plugin's `lus_open` function.

`ver` *QuotedString*: The version number of the plugin, as constructed from the first three elements of the array returned by the plugin's `lus_version` function.

`id` *QuotedString*: The 'ID' of the plugin, constructed from the fourth element and on of the array returned by the plugin's `lus_version` function.

`desc` *QuotedString*: A one line description of the plugin supplied by the plugin's `lus_one_line_description` function.

`checkBox`: An array element that describes a plugin checkbox. These elements are defined by the plugin populating `lus_CheckBoxInfo` structures in the `lus_GUI_Info_t` object that the plugin sends to the system using a call to the `SetLusGuiInfo` function.

Index: Unique index for each checkbox. Sequentially ascending integer.

Label: Brief description of the checkbox.

Tooltip: More extensive description.

EnableExpression: *Deprecated*. This is a boolean expression that may be used to determine if the check box is currently relevant (depends on other plugin field values).

<i>InitValue:</i>	A 1 (one) indicates the box is checked and a 0 (zero) indicates the box is unchecked.
<code>radiobuttons:</code>	An array of elements that describe the mutually exclusively set of radio buttons. These elements are defined by the plugin populating the <code>lus_RadioButtonInfo</code> structure in the <code>lus_GUI_Info_t</code> object that the plugin sends to the system using a call to the <code>SetLusGuiInfo</code> function.
<i>Label:</i>	The main descriptive label for the radio button box, if any. Supplied by the plugin's <code>lus_radiobox_label</code> function.
<i>Tooltip:</i>	The main tooltip help text for the radio button box, if any. Supplied by the plugin's <code>lus_radiobox_tooltip</code> function.
<i>EnableExpression:</i>	<i>Deprecated.</i> This is a boolean expression that may be used to determine if the radio button is currently relevant (depends on other plugin field values). Supplied by the plugin's <code>lus_radiobox_enable_exp</code> function.
<i>InitValue:</i>	The index (starting at zero) of the initially selected radio button. Supplied by the plugin's <code>lus_initial_radiobutton</code> function.
<i>ButtonLabels:</i>	
<i>Index:</i>	Unique index for each radio button. Sequentially ascending integers.
<i>Label:</i>	Brief description of the radio button.
<i>end:</i>	Marks the end of <i>ButtonLabels</i> .
<code>number:</code>	An array element that describes a plugin numeric data entry field. These elements are defined by the plugin populating <code>lus_NumControlInfo</code> structures in the <code>lus_GUI_Info_t</code> object that the plugin sends to the system using a call to the <code>SetLusGuiInfo</code> function.
<i>Index:</i>	Unique index for each number field. Sequentially ascending integer.
<i>Label:</i>	Brief description of the number field.
<i>Tooltip:</i>	More extensive description, such as range and semantics.
<i>EnableExpression:</i>	<i>Deprecated.</i> This is a boolean expression that may be used to determine if the number field is currently relevant (depends on other plugin field values).
<i>FieldType:</i>	<i>Deprecated.</i> Indicates suggested GUI input control to use. Possible values are: 0: Slider. 1: Spinner. 2: Simple.
<i>HardMin:</i>	The smallest valid input value.
<i>HardMax:</i>	The largest valid input value.

- InitMin*: *Deprecated*. Indicates a suggested minimum value to be used by any GUI control that is to automatically step through a series of values over a period of time.
- InitMax*: *Deprecated*. Indicates a suggested maximum value to be used by any GUI control that is to automatically step through a series of values over a period of time.
- InitValue*: The default value to be used on start or on plugin reset.

- **getplugin** *Flow Interface*

Returns the plugin's current state and field values for the given interface flow. The return value has the following format:

```

PluginSet := ["off" | "on"] ["init"] [CheckData] [ButtonData] [NumData]
             [ArgvData]
CheckData := {"check" | "uncheck"} Index [CheckBoxData]
ButtonData := "button" Index [ButtonData]
NumData := "num" Index uint32 [NumberData]
ArgvData := "argv" ArgumentList

```

- off: The plugin is disabled for the given interface flow.
- on: The plugin is enabled for the given interface flow.
- init: Any tokens that follow this are modifying the plugin from its initial state (as defined in the plugin's metadata).
- check *Index*: Checkbox *Index* is checked.
- uncheck *Index*: Checkbox *Index* is unchecked.
- button *Index*: Radio button *Index* is selected.
- num *Index uint32*: Numeric field *Index* has value *uint32*.
- argv *ArgumentList*: A list of zero or more argument tokens (quoted or unquoted). The meaning of the tokens is determined by the plugin.

- **setplugin** *Flow Interface PluginSet*

This command is used to set the plugin's state and field values. Syntax and semantics for *PluginSet* is described in the **getplugin** command above. However, the plugin is allowed to return additional arguments after the Okay and Fail responses. The number and meaning of these additional arguments is defined by the plugin.

- **resetplugin** *Flow Interface*

Resets the plugin field settings to their initial values for the given interface flow.

Statistics

- **zerostats** *Flow Interface*

Set the statistic counter values to zero for the given interface flow.

- **getstats** "info" | {*Flow Interface* ["all" | "trigger"]}

If “info” is supplied then three strings are returned for each statistics field (so the number of strings returned is always a multiple of three.) The fields returned are an ID field unique to each statistic field (suitable for use as a variable name in most computer languages), a short label string suitable as a row or column header, and a longer descriptive string suitable as a GUI tool tip help. The formal syntax for the return value of the “info” variant is:

```
StatInfo :=      ID Label ToolTip [StatInfo]
```

If a flow and interface number are supplied along with the token “trigger”, then a count of the triggers pending, triggered impairments started, and triggered impairments completed (in that order) on each trigger type is returned. The formal syntax for the return value of the “trigger” variant is:

```
TriggerInfo :=  "basic" Cnts "tcpv4" Cnts "udpv4" Cnts "tcpv6" Cnts "udpv6" Cnts  
Cnts :=        uint32 uint32 uint32
```

If a flow and interface number are supplied along with the token “all”, then the ID field and corresponding counter value are returned for all the non-trigger-related statistic fields; that is, any new standard impairment statistics fields and any plugin created entries. If only a flow and interface number are supplied (no “all” or “trigger”), then a “classic” subset of the non-trigger-related statistic fields are returned. The formal syntax for the return value of the “all” variant is:

```
Stats :=        ID uint32 [Stats]
```

The following IDs constitute the “classic” subset that are returned when no “all” is included:

- in *uint32*: The number of incoming packets that met the match criteria for the interface flow.
- out *uint32*: The number of outgoing packets from the interface flow.
- dropped *uint32*: The number of incoming packets that met the match criteria that were subsequently dropped (not forwarded).
- coupleddropped *uint32*:
The number of incoming packets that met the match criteria that were subsequently dropped (not forwarded) because they were coupled to previously dropped packets.
- duped *uint32*: The number of incoming packets that met the match criteria that were subsequently duplicated (forwarded more than once).
- coupleduped *uint32*:
The number of incoming packets that met the match criteria that were subsequently duplicated (forwarded more than once) because they were coupled to previously duplicated packets.
- jittered *uint32*: The number of incoming packets that met the match criteria that were subsequently jitter delayed (forwarded after a variable delay).
- couplejittered *uint32*:

The number of incoming packets that met the match criteria that were subsequently jitter delayed (forwarded after a variable delay) because they were coupled to previously delayed packets.

`reordered uint32`: The number of incoming packets that met the match criteria that were subsequently jitter delayed (forwarded after a variable delay) and were sent out after subsequent packets passed them.

Log File Control

- `getlogfile`
Returns the full path name of the log file, if any.
- `getlogging Flow Interface`
Gets the current log flags that describe the traffic detail that is being stored into the log file for the given interface flow. The syntax and semantics of the return value is the same as that described for the `LogFlags` element of the `setlogging` command.
- `setlogging Flow Interface LogFlags`
Sets the amount of detail to be included in the packet traffic log file for the given interface flow.

```
LogFlags := {"clear" | "in" | "out" | "details_in" | "details_out" |  
"settings" | "fate" | "stats"} [LogFlags]
```

The flags are:

<code>clear:</code>	Do not log anything. May be followed by flags that turn logging back on.
<code>in:</code>	Log basic information on each incoming packet.
<code>out:</code>	Log basic information on each outgoing packet.
<code>details_in:</code>	Log detailed information on each incoming packet.
<code>details_out:</code>	Log detailed information on each outgoing packet.
<code>settings:</code>	Log server setting changes.
<code>fate:</code>	Log impairments applied to each packet.
<code>stats:</code>	Periodically log statistic counts.

Miscellaneous Commands

- `useversion 2`
The Standard Impairment server is designed to provide backward support for previous versions of the command language. This means an end user is not forced to rewrite older external scripts with each upgrade of Maxwell's command language. However, newer versions of the server always default to the latest version of the language. So to insure that older scripts continue to work, the first command they should issue to the server is the

useversion command, giving the version number the scripts were written against. The version described in this document is version 2.

Note that useversion does not exist for shell commands since they run independent of each other and therefore retain no common state.

- **getnumflows**
Returns the maximum number of flows that may be provisioned on each interface.
- **clearallflows**
Resets statistic counts to zero and match criteria, impairments, and plugin field values (if any) to their initial values for all flows on both interfaces.
- **getstartargs**
Returns a string containing the series of command line arguments that would need to be used to start a different stdiserver process with the same state as the currently running stdiserver process. Often used in conjunction with the restartserver command.
- **restartserver** [*ArgumentList*]
Using the exec() system call, restarts the stdiserver process "in place" and uses the tokens in *ArgumentList*, if any, as the new set of command line arguments to the new process image. This command provides a way to remotely change the plugin, number of flows, and other aspects that are set only at startup.
- **setmaxtap** {**addrnet4** *IPv4Address/uint8* | **addrnet6** *IPv6Address/uint8* | **route4** *IPv4Address [IPv4Address]* | **route6** *IPv6Address [IPv6Address]*}
The **addrnet4** and **addrnet6** forms are used to set the address and netmask for the maxtap interface device for IPv4 and IPv6 respectively. The **route4** and **route6** forms are used to assign IPv4 and IPv6 static routes to the maxtap interface. The second optional address is deleted from the static route table (assuming it exists).
- **debugflags** [*uint32*]
When supplied with an integer argument, this turns some simple debug output of the Remote API handler inside the stdiserver program. Currently only two bits are used: when bit 0 is set the handler echoes every command received by it from clients and when bit 1 is set it prints out a notice whenever any client connects or disconnects from the handler.
- **Getifinfo** *Interface*
Returns the name (e.g. eth1, eth2) , current connection state, negotiated speed, and negotiated duplex of the requested interface. The syntax of the response is:

```
IfInfo := name {"connected"|"unconnected"} "speed" uint32"Mbps"  
         "duplex" {"full"|"half"}
```

BNF Summary for the Standard Impairment Server CLI

```
Command := "useversion" 2 |
           "getpluginmeta" |
           "getplugin" Flow Interface |
           "setplugin" Flow Interface PluginSet |
           "resetplugin" Flow Interface |
           "getnumflows" |
           "clearallflows" |
           "setmatch" Flow Interface ["zerostats"] Match |
           "setImpair" Flow Interface ["zerostats"] Impairment |
           "zerostats" Flow Interface |
           "getmatch" Flow Interface |
           "getimpair" Flow Interface |
           "setlogging" Flow Interface LogFlags |
           "getlogging" Flow Interface |
           "getstats" Flow Interface |
           "getifinfo" Interface |
           "getlogfile"

Response := { RspFail | "Okay" PluginMeta } | # Response to getpluginmeta.
           { RspFail | "Okay" PluginSet } | # Response to getplugin.
           { RspFail | "Okay" [ArgumentList] } | # Response to setplugin.
           { RspFail | "Okay" uint32 } | # Response to getnumflows.
           { RspFail | "Okay" Match } | # Response to getmatch.
           { RspFail | "Okay" Impairment } | # Response to getimpair.
           { RspFail | "Okay" LogFlags } | # Response to getlogging.
           { RspFail | "Okay" Stats } | # Response to getstats.
           { RspFail | "Okay" QuotedString } | # Response to getlogfile.
           { RspFail | "Okay" IfInfo } | # Response to getifinfo.
           { RspFail | "Okay" } # Response to all other commands.

RspFail := "Fail" Description
Match := "none" | "all" |
        ["allbut"] ["bytes" Bytes] ["lan" Lan] ["ipv4" IPv4 | "ipv6" IPv6]
        ["trigger" {"basic" | "tcpudp"} uint32 uint32 uint32]

Lan := {"usesvlan" |
        "vlanid" uint16 Uint16Mask |
        "priority" uint8 Uint8Mask |
        "type" uint16 Uint16Mask |
        "addr" SourceOrDest MacAddress MacAddressMask} [Lan]

IPv4 := {"port" SourceOrDest uint16 Uint16Mask |
        "addr" SourceOrDest IPv4Address IPv4AddressMask |
        "qos" uint8 Uint8Mask |
        "flags" uint8 Uint8Mask |
        "proto" uint8 Uint8Mask} [IPv4]

IPv6 := {"port" SourceOrDest uint16 Uint16Mask |
        "addr" SourceOrDest IPv6Address IPv6AddressMask |
        "class" uint8 Uint8Mask |
        "flow" uint32 Uint32Mask |
        "proto" uint8 Uint8Mask } [IPv6]

Bytes := OffsetBase Offset uint32 Uint32Mask [Bytes]
Impairment := "forwardAll" | "dropall" | [Drop] [Dup] [Delay] [Rate] [Jitter] [Corrupt]
Drop := "drop" percent ["couple" percent CouplingInterval ["slew"]]
Dup := "dup" percent ["couple" percent]
Delay := "delay" uint32
```

```

Rate :=      "rate" XmitRate MinPayload MaxPayload Overhead QLength
             ActualRate ActualDuration
Jitter :=    "jitter" percent JitterTime ["reorderok"]
             ["couple" percent CouplingInterval ["slew"]]
Corrupt :=   "corrupt" ["payload"] percent
JitterTime := "fixed" uint32 |
             "linear" LowerBound UpperBound |
             "gaussian" Mean StdDeviation |
             "piecewise" Pieces
Pieces :=    "end" | Piece Pieces
Piece :=     PieceWidth {"fixed" uint32 | "linear" LowerBound UpperBound |
             "gaussian" Mean StdDeviation}
LogFlags :=  {"clear" | "in" | "out" | "details_in" | "details_out" | "settings" |
             "fate" | "stats"} [LogFlags]
PluginSet := ["off" | "on"] ["init"] [CheckData] [ButtonData] [NumData] [ArgvData]
CheckData := {"check" | "uncheck"} [CheckData]
ButtonData := "button" Index [ButtonData]
NumData :=   "num" Index uint32 [NumData]
ArgvData :=  "argv" ArgumentList
PluginMeta := "path" QuotedString "opts" QuotedString "ver" QuotedString
             "id" QuotedString "desc" QuotedString
             [CheckBoxes] [RadioButtons] [NumberFields]
CheckBoxes := "checkbox" Index Label Tooltip EnableExpression InitValue
             [CheckBoxes]
RadioButtons := "radiobuttons" Label Tooltip EnableExpression
             InitValue ButtonLabels
ButtonLabels := "end" | Index Label ButtonLabels
NumberFields := "number" Index Label Tooltip EnableExpression FieldType
             HardMin HardMax InitMin InitMax InitValue [NumberFields]
StatInfo :=  ID Label ToolTip [StatInfo]
Stats :=     ID unit32 [Stats]
IfInfo :=    token {"connected" | "unconnected"} "speed" uint32 "Mbps"
             "duplex" {"full" | "half"}
TriggerInfo := "basic" Cnts "tcpv4" Cnts "updv4" Cnts "tcpv6" Cnts "udpv6" Cnts
Cnts :=      uint32 uint32 uint32
ArgumentList := {QuotedString | token} [ArgumentList]
ActualDuration := uint32
ActualRate :=     uint32
CouplingInterval := uint32           # In microseconds.
Description :=    QuotedString       # Describes the nature of the failure.
EnableExpression := QuotedString
FieldType :=      "0" | "1" | "2"    # Types are slider (0), spinner (1), and simple (2).
Flow :=          uint32              # Identifies a flow and its match order.
HardMax :=       uint32
HardMin :=       uint32
ID :=            QuotedString        # ID or variable name usable in most languages.
Index :=         uint32
InitMax :=       uint32
InitMin :=       uint32
InitValue :=     uint32
Interface :=     "0" | "1"          # Maxwell Ethernet interface port number.
IPv4AddressMask := IPv4Address
IPv6AddressMask := IPv6Address
Label :=         QuotedString
LowerBound :=    uint32              # Smallest random number to return.
MacAddress :=    OctetList           # Six octet value of the Ethernet address.
MacAddressMask := MacAddress

```

```

MaxPayload :=      uint32
Mean :=           uint32           # Mean of the random numbers.
MinPayload :=     uint32
Name :=          token
Offset :=        uint16
OffsetBase :=    "frame" | "framedata" | "ipdata" | "tcpudpdata"
Overhead :=      uint32
PieceWidth :=   uint16
QLength :=      uint32
SourceOrDest :=  "src" | "dst" | "both" | "notused"
StdDeviation :=  uint32           # Standard deviation of the random numbers.
ToolTip :=      QuotedString
Uint8Mask :=     uint8
Uint16Mask :=   uint16
Uint32Mask :=   uint32
UpperBound :=   uint32           # Largest random number to return.
XmitRate :=     uint32

```

Basic token types (the “uint” strings may be in decimal (e.g. 42), octal (e.g. 077), or hexadecimal (e.g. 0x1F) notation):

- token Any sequence of characters.
- uint8 A string representing a value from 0 to 255.
- uint16 A string representing a value from 0 to 65535.
- uint32 A string representing a value from 0 to 4294967295.
- percent A string representing a floating point value from 0.0 to 100.0.
- IPv4Address IPv4 notation (e.g. 127.0.0.1).
- IPv6Address IPv6 notation (e.g. 0:0:0:0:0:0:A00:1).
- OctetList An even number of hexadecimal digits (e.g. 0ABF898F000010).
- QuotedString A sequence of characters between double quotes (“”).

The Perl API

If you wish to control and monitor Maxwell with Perl scripts, you have two options: write your own code that writes text commands to the Remote API port and parses the resulting responses, or use the Perl module StdServer that is supplied by InterWorking Labs. The source code for that module is available on the Maxwell machine as this file:

```
/usr/local/iwl/src/StdServer.pm
```

You should be able to copy that file to any platform that has Perl installed - and then control Maxwell from there. The functions in the StdServer.pm module are described in that file and you should use it as primary documentation. Additional detail on the proper usage of the module and its functions may be obtained by studying the following usage examples, also stored on Maxwell:

```
/usr/local/iwl/src/example1.pl      Simple impairment.
/usr/local/iwl/src/example2.pl      Simple flow match and impairment.
/usr/local/iwl/src/example3.pl      More complex flow match and more complex impairment.
/usr/local/iwl/src/example4.pl      Controlling a plugin.
/usr/local/iwl/src/example_tcp_plugin.pl Running an impairment test in TCPIP plugin
/usr/local/iwl/src/example_sip_plugin.pl Running an impairment test in SIP plugin
```

In all cases, if an error is returned by the server or an error occurs in the StdServer module, the functions will terminate by issuing a call to "croak".

The Java API

You can use Maxwell Java API to control the Maxwell Standard Impairment Server from any platform that Java supports. The Java API documentation was generated using Javadoc and the resulting class information is stored at this location on Maxwell:

```
/usr/local/iwl/stdijava_api/javadoc/index.html
```

The Jar file is located here:

```
/usr/local/iwl/stdijava_api/stdijava_api.jar
```

An example program that uses the Java API is located here:

```
/usr/local/iwl/stdijava_api/Example1.java
```

To compile and run the example, the following steps need to be done:

1. If you do not already have it installed on your target machine, download and install Java SDK at <http://java.sun.com/javase/downloads/index.jsp>
2. Make sure stdserver is running on Maxwell.

3. Copy the file `/usr/local/iwl/stdijava_api/stdijava_api.jar` from Maxwell to a directory on the machine you will be using the Maxwell Java API. For example:
`/home/bob/maxjavaapi` or `c:\maxjavapi`.
4. Find the sample Java program shipped with Maxwell found at `/usr/local/iwl/stdijava_api/Example1.java`
(This is a basic example showing basic usage.)
Copy the sample program to a user directory on the machine you will be using Maxwell Java API. For example:
`/home/bob/maxtest` or `c:\maxtest`
5. Set `CLASSPATH` environment variable properly on Unix and Linux Systems. For example, in c-shell you would do this:
`setenv CLASSPATH ./home/bob/maxjavaapi/stdijava_api.jar:/home/bob/maxtest`
On Windows you need to do this:
In Control Panel->System->Advanced->Environment Variables, set:
`CLASSPATH .C:\maxjavaapi\stdijava_api.jar;c:\maxtest`
6. Compile and run the sample Java program. For example:
`cd c:\maxtest`
`javac Example1.java`
`java Example1`
7. Use the `getplugin`, `getmatch`, and `getimpair` shell commands to validate that any changes attempted by the example program were successful.

The Python API

If you wish to control and monitor Maxwell with Python programs, you have two options: write your own code that writes text commands to the Remote API port and parses the resulting responses, or use the Python module `servercomm` that is supplied by InterWorking Labs. The source code for that module is available on the Maxwell machine as this file:

```
/usr/local/iwl/src/servercomm.py
```

You should be able to copy that file to any platform that has Python installed - and then control Maxwell from there. Additional detail on the proper usage of the module and its functions may be obtained by studying the source code of the module itself and the usage examples, also stored on Maxwell.